AFFORDCODE: LEARNING GENERALIZABLE AFFORDANCES AS EXECUTABLE PROGRAMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Robotic affordance learning treats affordances as prediction outputs, including interaction regions, poses, or trajectories. Advances in language models and programmatic object modeling enable inspectable, editable, and composable affordance programs, whose potential for feedback-driven refinement and cross-instance reuse remains underexplored. To bridge this gap, we present AffordCode, an agentic framework that learns reusable affordance programs over parameterized object templates exposing part geometry and composition. Following inter-part structural relationships, AffordCode constructs functional chains bottom-up, composing part-level functions into object-level programs that compute instance-specific affordances. A closed-loop procedure verifies candidate programs, traces structural, geometric, and interaction feedback to faulty assumptions and dependencies, and refines the corresponding code. This method offers three advantages: (1) Precise Feedback-Driven Refinement where editable code exposes faulty interaction logic; (2) Coherent Object-Level Behavior since bottom-up composition preserves part dependencies and aligns local functions with the task; and (3) Synthesis-Free Reuse: validated part functions can be recomposed and object programs instantiated across compatible objects. Extensive experiments demonstrate the effectiveness of the generated affordances, their generalization across object instances, and their utility in downstream tasks.

1 INTRODUCTION

Robotic affordance learning concerns how an agent can interact with objects and what effects these interactions can produce. A classical formulation describes affordances through relations among entities, behaviors, and effects (Sahin et al., 2007). Existing approaches operationalize these relations through actionable parts and poses (Geng et al., 2022), local actionability and interaction proposals (Mo et al., 2021), and motion fields (Eisner et al., 2022). These representations provide useful spatial and functional information for interaction. Learning the underlying rules that connect object structure to these outputs offers a way to make this information inspectable, adaptable, and reusable. Recent advances in language models and programmatic object modeling are making programmatic object representations increasingly accessible. CAD-Recode reconstructs executable CAD code from point clouds (Rukhovich et al., 2025), while Real2Code recovers articulated object models through code generation (Mandi et al., 2024). Articulate-Anything further demonstrates how vision-language agents can generate and iteratively refine programs for articulated object modeling (Le et al., 2025). These developments provide increasingly practical means of obtaining object descriptions with explicit geometric parameters and structural relationships. At the same time, ConceptFactory and Digital Gene demonstrate how parameterized object templates can support procedural definitions and transfer of object knowledge, including affordances (Sun et al., 2024; Sun & Lu, 2025). Together, these advances motivate using programmatic object representations as a foundation for learning affordances in code.

Code offers an explicit, editable, and composable representation of the interaction rules underlying affordance outputs. A program can query object parts, evaluate geometric relationships, and combine local functions into an interaction. For example, a drawer-opening program can connect the handle's contact geometry to the drawer's motion constraints and recompute the corresponding poses and motion as the object's parameters change. Language-model-generated policies also demonstrate the potential of composing perception and control routines in code (Liang et al., 2023).

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.

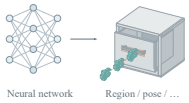
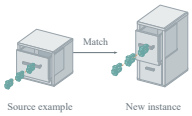
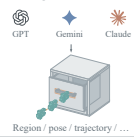
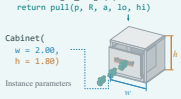
Affordance acquisition	Affordance prediction Where2Act / FlowBot3D	Affordance transfer AffordDP / AffordGen	VLM reasoning Direct inference	Program synthesis AffordCode (ours)
Example Open a drawer				
Can interaction errors be corrected locally?	— No directly editable interaction rules	✓ Editable examples and transfer parameters	— No direct control over internal rules	✓ Targeted edits to interaction programs
Can it learn from interaction feedback?	✓ Learning through parameter updates	— No feedback-driven affordance refinement	— No interaction-learning mechanism	✓ Feedback-driven program refinement
Can learned rules support further affordance learning?	— No explicit rule-reuse mechanism	— Transfers examples without learning reusable rules	— No persistent learned-rule library	✓ Reuse validated rules to learn new affordances
Can it support new affordance outputs?	— Restricted to predefined outputs	— Restricted to source representations	✓ Flexible output types through prompting	✓ New affordance outputs through structural reasoning and program composition

Figure 1: **Comparison of affordance acquisition strategies.** AffordCode learns executable interaction rules that support refinement and reuse. Check marks denote explicit support in the illustrated workflows; dashes denote mechanisms not included.

However, the potential of code for systematically acquiring and improving reusable affordance programs remains underexplored. Learning such programs requires organizing part-level functions around object-level interaction goals and translating verification feedback into corrections to the underlying interaction rules.

To address this challenge, we present **AffordCode**, an agentic framework that learns affordances as reusable programs over parameterized object templates. Given a task description and an object representation exposing parts, geometry, poses, and composition, the agent constructs affordance programs from the bottom up. It retrieves or synthesizes part-level functions, binds them to the relevant parts, and composes them according to the object's structural relationships. The resulting functional chains make the dependencies among participating parts explicit and align their local functions with the object-level interaction goal. Evaluating these programs with instance-specific template parameters produces the requested affordance outputs, such as interaction regions, poses, or trajectories. Figure 1 compares AffordCode with representative affordance prediction, example transfer, and direct VLM inference workflows.

AffordCode improves candidate programs through a closed-loop procedure for verification, diagnosis, and refinement. Structural, geometric, and interaction feedback assesses program validity and the consistency of generated outputs with the task, supported by visualizations of the parameterized object and its affordance outputs. The agent uses explicit program dependencies to trace problematic outputs to the relevant functions and assumptions, revise the implicated code, and re-evaluate the program. A hierarchical code memory retains validated part-level functions and object-level programs, incorporating accepted revisions for subsequent retrieval, composition, and instantiation.

This design offers three advantages. First, interaction feedback can directly inform targeted refinement because the underlying interaction logic is explicit and editable in code. Second, bottom-up synthesis constructs object-level functionality incrementally from local part functions while preserving the dependencies required for the overall interaction goal. Third, validated part-level functions can support new compositions, while object-level programs can generate outputs for compatible instances by evaluating new template parameters without renewed synthesis. We evaluate AffordCode in terms of affordance accuracy, generalization across object instances, and utility in downstream tasks.

Our contributions are as follows:

- We present AffordCode, an agentic framework that learns affordances as reusable programs grounded in parameterized object templates. The framework unlocks the potential of code as an explicit, editable, and composable affordance representation, enabling flexible affordance generation across object instances.

- We develop a bottom-up synthesis approach for affordance programs guided by object structure. This enables compositional reasoning from local part functions to object-level functionality, supporting effective interactions aligned with task goals.
- We develop a closed-loop procedure for verifying and refining affordance programs. This enables feedback-driven self-correction of interaction logic, progressively improving its consistency with object geometry and task requirements.
- We present a comprehensive experimental evaluation of AffordCode, assessing the effectiveness of its generated affordances, generalization across object instances, and utility in downstream tasks.

2 Related Work

Affordance representations. Robotic affordance representations connect object observations or structure to possible interactions. Part-centric approaches such as GAPartNet (Geng et al., 2022) associate semantic part categories and canonical poses with transferable manipulation strategies. Interaction-centric approaches use local predictions: Where2Act (Mo et al., 2021) predicts actionability and interaction proposals, while VAT-Mart (Wu et al., 2022) learns task-conditioned actionability and trajectories. FlowBot3D (Eisner et al., 2022) represents articulation through dense pointwise motion directions for downstream planning. Together, these formulations capture functional parts, contact choices, and motion.

Programmatic object representations. Programmatic representations encode object geometry and structure as parameterized, executable models. ConceptFactory (Sun et al., 2024) organizes objects with geometric templates and defines knowledge through procedures such as region membership functions and pose transformations. Other methods automate object-model acquisition: CAD-Recode (Rukhovich et al., 2025) reconstructs CAD programs from point clouds, while Real2Code (Mandi et al., 2024) reconstructs articulated objects through code generation. Articulate-Anything (Le et al., 2025) uses vision-language agents and iterative visual feedback to generate and refine articulation programs. These studies provide foundations for programmatic object modeling and associating functional knowledge with object parameters and structure.

Program generation, refinement, and reuse. Language agents use programs and executable objectives to connect task descriptions with interaction. Code as Policies (Liang et al., 2023) generates hierarchical policies from perception and control APIs, while RoboScript (Chen et al., 2024) integrates generated code with perception and motion planning. Related methods formulate tasks as optimization objectives: VoxPoser (Huang et al., 2023) constructs observation-grounded value maps, ReKep (Huang et al., 2024b) generates executable keypoint constraints, and CoPa (Huang et al., 2024a) specifies spatial constraints over object parts. Feedback and memory enable further improvement: Reflexion (Shinn et al., 2023) retains linguistic reflections from task feedback, while Voyager (Wang et al., 2023) refines action programs using environment feedback, execution errors, and self-verification, storing successful programs for reuse. AffordCode instead acquires and refines affordance programs over parameterized objects through structure-guided composition and feedback-driven revision of part-level and object-level interaction logic.

3 Method

AffordCode learns affordances by acquiring reusable programs over programmatic object templates. Given an object description and a task, a coding agent retrieves relevant code and composes part-level computations into object-level interaction logic. Candidate programs are verified, diagnosed, and refined using structured feedback. Validated code is retained in a hierarchical memory and evaluated with new instance parameters to produce the corresponding affordance outputs.

3.1 Executable Affordance Representation

AffordCode represents an affordance as an executable program that maps parameterized object structure to task-specific interaction quantities.

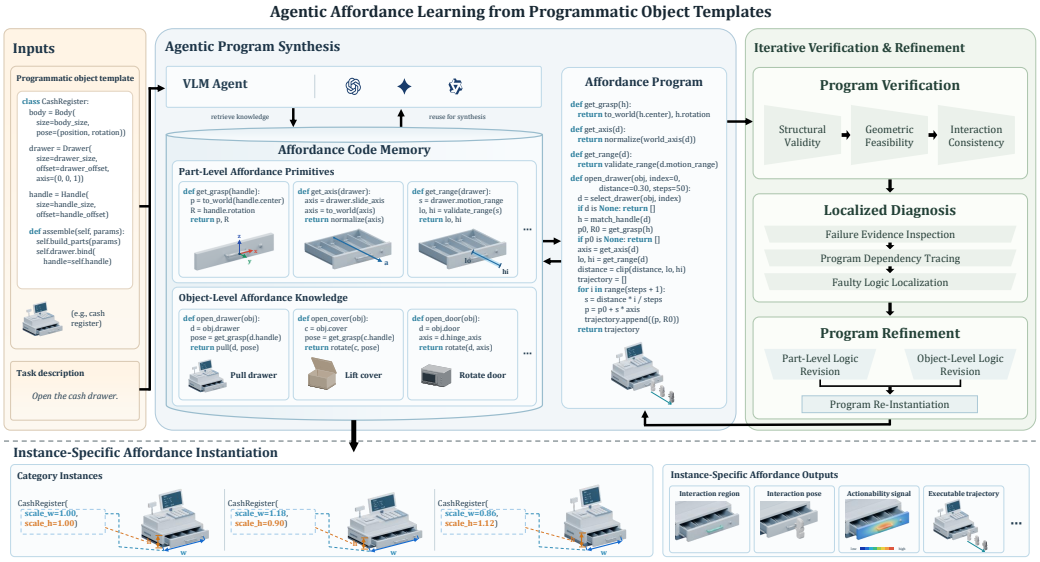


Figure 2: Overview of **AffordCode** framework. Given a programmatic object template and task description, a VLM-based agent retrieves reusable code and synthesizes an affordance program with part-level computations and object-level interaction logic. The program computes interaction quantities from object structure and undergoes task-specific structural, geometric, and interaction checks, using rendered views and candidate outputs as visual feedback for verification and diagnosis. Failed checks trigger targeted refinement at the part level, object level, or both, followed by re-instantiation and verification. Verified programs are retained in memory and instantiated on compatible objects to produce task-specified interaction regions, poses, actionability signals, and trajectories.

Inputs and objective. Given a programmatic object representation $\mathcal{G}_O$, its template definitions, and a task t, AffordCode acquires an affordance program reusable across compatible object instances. $\mathcal{G}_O$ exposes object parts, geometry, poses, parameters, and composition, while t specifies the intended interaction and required outputs.

Part-level and object-level programs. For object category c and a fixed task t, whose dependence is omitted for clarity, the affordance program is

$$\mathcal{A}_c = (\mathcal{P}_c, \mathcal{O}_c)\,, \tag{1}$$

where $\mathcal{P}_c$ contains reusable part-level routines and $\mathcal{O}_c$ composes them into object-level interaction logic. Part-level routines compute local quantities, such as interaction regions, contact frames, motion axes, and admissible ranges, from part geometry and poses. Object-level routines bind participating parts and combine their computations according to object structure and the task goal.

Instance-specific outputs. Executing the program on an object instance yields

$$y_O = \mathrm{Eval}\,(\mathcal{A}_c, \mathcal{G}_O)\,, \tag{2}$$

where y_O may be an interaction region, contact pose, applicability condition, or motion sequence. Because y_O is computed from $\mathcal{G}_O$, the same program adapts to compatible object instances without renewed synthesis. **Additional specifications and examples are provided in Appendix A.2.**

3.2 Knowledge-Guided Program Synthesis

Hierarchical code memory. AffordCode maintains a hierarchical memory

$$\mathcal{M} = (\mathcal{M}_{\text{part}}, \mathcal{M}_{\text{object}})\,, \tag{3}$$

where $\mathcal{M}_{\text{part}}$ stores reusable part-level routines from object templates and previously generated programs, while $\mathcal{M}_{\text{object}}$ stores validated object-level programs and their part-level dependencies.

Each entry records its functional role, applicable templates, input–output interfaces, and validation evidence.

Retrieval jointly considers the task, object category, and part-template composition. The agent first searches for a compatible object-level program and retrieves its dependencies. If no complete match exists, it retrieves relevant part-level routines and related object-level programs for composition. The retrieved code forms the synthesis context $\mathcal{K}$.

Grounding and composition. The agent binds functional roles to object parts and checks the required entities, geometric relationships, and routine assumptions. A compatible object-level program is instantiated directly; otherwise, the agent composes retrieved routines and implements missing computations. The resulting program preserves the dependencies connecting object parts, interaction quantities, and the requested output.

Task-specified outputs. The task description and optional examples determine which quantities the program exposes. Existing part grounding and interaction computations are reused when extending the program to new output forms. **Implementation details for program synthesis are provided in Appendix A.3.**

3.3 Verification, Diagnosis, and Refinement

For each template–task pair, AffordCode verifies candidate programs on parameterized instances sampled from the template's valid ranges. These instances preserve part composition while varying dimensions, poses, and other parameters. They are independent of downstream evaluation instances and are used only to test whether the interaction logic generalizes across compatible parameterizations.

Program verification. Each candidate undergoes three complementary checks. *Structural validity* verifies program execution, part references, routine interfaces, output types, and coordinate frames. *Geometric feasibility* checks whether the generated quantities agree with the instantiated object, including part membership, contact compatibility, collision, and clearance. As shown in Figure 3, rendered candidate outputs provide visual feedback that exposes spatial and functional discrepancies and guides revision of the responsible interaction logic.

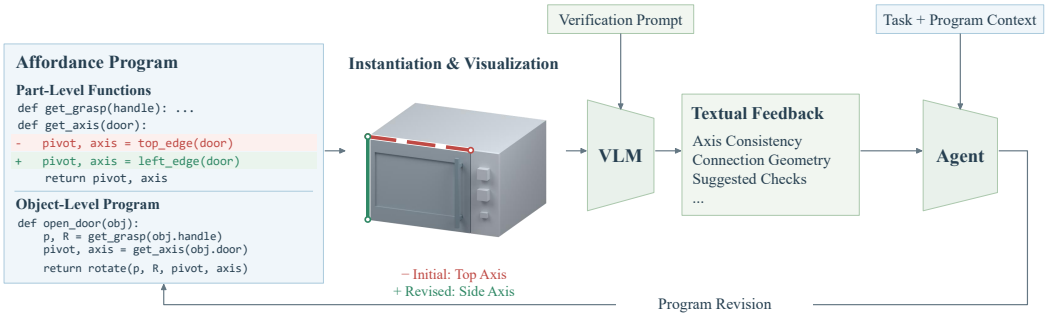


Figure 3: **Visual prompting for program refinement.** Rendered candidate outputs expose discrepancies and guide targeted code revision.

Interaction consistency evaluates whether the generated interaction satisfies the task constraints and intended effect. Given an initial state x_0 and interaction u, we obtain

$$\hat{x}_{1:H} = \text{Rollout}\left(x_0, u; \mathcal{G}_O\right), \tag{4}$$

and require

$$C_t(\hat{x}_{0:H}) = 1, \qquad \Phi_t(x_0, \hat{x}_H) = 1, \tag{5}$$

where C_t denotes the interaction constraints and Φ_t the target effect. The task condition remains fixed throughout refinement. **Detailed checks are provided in Appendix A.4.**

Localized diagnosis. A failed check produces structured feedback $\mathcal{F}^{(r)}$ describing the violated condition, involved entity, observed discrepancy, and relevant intermediate values. Using program dependencies, the agent traces the failure to the responsible assumption, expression, or routine, localizing the required revision to the part level, object level, or both.

Program refinement. At round r, the candidate is updated as

$$\mathcal{A}^{(r+1)} = \mathrm{Revise}\left(\mathcal{A}^{(r)}, \mathrm{Diagnose}\left(\mathcal{A}^{(r)}, \mathcal{F}^{(r)}\right), \mathcal{K}\right), \tag{6}$$

where $\mathcal{K}$ is the retrieved synthesis context. Errors in reusable part computations trigger part-level revisions, whereas errors in entity association, composition, or task-specific logic trigger object-level revisions. The revised program is re-instantiated and re-evaluated while unrelated validated code is preserved. This process continues until all required checks pass or the refinement budget is exhausted. Only verified programs are consolidated into the hierarchical code memory, where they can be retrieved and instantiated on compatible object instances or reused as context for subsequent affordance acquisition. Details of knowledge consolidation and reuse are provided in Appendix A.5.

4 Experiments

We evaluate AffordCode along five complementary dimensions:

Can executable affordance programs support successful manipulation when perception errors are removed (Section 4.1)? Can AffordCode perform diverse end-to-end manipulation tasks from RGB-D observations (Section 4.2)? Can program-generated demonstrations improve downstream policy learning and generalization (Section 4.3)? Can AffordCode guide a frozen policy at inference time, and how much do program verification and bottom-up composition contribute (Section 4.4)? Can AffordCode produce effective affordance outputs on real-world objects (Section 4.5)?

Affordance acquisition protocol. We use GPT-5.6 Sol throughout all experiments. Unless otherwise specified, each template–task program is acquired using eight randomly parameterized instances and at most three verification-and-refinement rounds before evaluation. The resulting program and code memory are then frozen; no evaluation instance or task-success signal is used for further refinement.

4.1 From Affordance Representations to Manipulation

Evaluation Setup. We construct a GAPartNet-derived evaluation (Geng et al., 2022) with 479 interactions across seven object families: doors, drawers, screens, covers, liftable lids, handles, and knobs. Each case requires a Panda robot to physically complete the target interaction. To isolate affordance-to-action conversion from perception, all methods receive representation-specific ground-truth inputs.

Baselines. *GAPartNet-GT* receives ground-truth part categories, segmentation, target-part bounding box, pose, and mesh (Geng et al., 2022); *Where2Act-GT* receives the interaction point, target mesh, and joint kinematics (Mo et al., 2021); *FlowBot3D-GT* receives the point cloud, target-part mask, target link, and joint information used to compute articulation flow (Eisner et al., 2022). *AffordCode* receives the ground-truth ConceptFactory code-form asset containing parts, geometry, parameters, and structural relations. No method receives a successful robot trajectory or motion plan.

Evaluation Metrics. We report per-family and overall interaction success rate (SR). Following GAPartNet (Geng et al., 2022), success requires the target part to reach at least 90% of the required motion range within 1,000 simulation steps.

Results. As shown in Table 1, AffordCode achieves an overall success rate of 42.80%, outperforming the strongest GT baseline, GAPartNet-GT, by 15.87 percentage points. These results demonstrate that executable affordance programs can more effectively translate structured object information into successful robot interactions.

Table 1: Interaction success rates (%) on 479 GAPartNet-derived cases with representation-specific ground-truth inputs. Header counts denote evaluated cases; bold denotes the best result.

Method	Door 138	Drawer 133	Screen 44	Cover 78	Lift lid 20	Handle 29	Knob 37	Overall 479
GAPartNet-GT	18.84	48.12	4.55	**32.05**	0.00	37.93	2.70	26.93
Where2Act-GT	16.67	49.62	6.82	24.36	0.00	44.83	2.70	26.10
FlowBot3D-GT	13.04	16.54	2.27	19.23	0.00	27.59	2.70	13.57
AffordCode	**29.71**	**62.41**	**79.55**	29.49	**5.00**	**48.28**	**21.62**	**42.80**

4.2 System-Level Manipulation on RoboTwin

Evaluation Setup. We evaluate AffordCode on RoboTwin (Chen et al., 2025) to assess system-level manipulation across diverse tasks. We select 42 tasks that do not require coordinated bimanual manipulation, covering object articulation, tool use, functional placement, and sorting and stacking. All methods receive RGB-D observations and task instructions and use the Aloha-AgileX robot under the `demo_clean` setting, with 20 matched episodes per task.

For AffordCode, a template-conditioned frontend grounds semantic parts in RGB, estimates observable geometry from depth, and infers the remaining template parameters. Implementation details and complete 42-task results are provided in Appendix A.6.

Baselines. We compare AffordCode with representative systems that translate visual observations and task instructions into robot actions. *VoxPoser* performs manipulation planning through 3D value maps (Huang et al., 2023); *CoPa* reasons over spatial constraints between functional parts (Huang et al., 2024a); *RoboScript* generates task-level robot programs (Chen et al., 2024); *ReKep* uses executable constraints over semantic keypoints (Huang et al., 2024b); and *MOKA* predicts keypoint affordances for contact and motion (Fang et al., 2024).

We additionally report the RoboTwin Expert Generator as a privileged reference using its officially reported results. Unlike the observation-based methods, the expert-generation pipeline leverages direct task and simulator information, including object annotations, to construct robot code and supports simulation-in-the-loop execution feedback and correction. We therefore treat it as a privileged reference rather than a directly matched baseline.

Evaluation Metrics. We report average task success rate, the number of tasks with nonzero success, and their conditional average success rate $\mathrm{SR}_{>0}$.

Results. As shown in Table 2, AffordCode achieves 36.67% average success, compared with 19.76% for the strongest observation-based baseline, ReKep, and succeeds on 35 of 42 tasks, indicating substantially broader task coverage than the comparison methods.

The privileged RoboTwin Expert Generator achieves 43.71% average success by leveraging direct task and simulator information together with expert code generation and execution feedback. In comparison, AffordCode reaches 36.67% while operating only from RGB-D observations and task instructions, with-

Table 2: RoboTwin manipulation results (%). Observation-based methods use 20 episodes per task; RoboTwin Expert is a privileged reference.

Task	VoxPoser	CoPa	Robo Script	ReKep	MOKA	**Afford Code**	RoboTwin Exp.
Beat Block Hammer	0	0	0	35	0	10	53
Blocks Ranking RGB	5	0	0	15	0	60	80
Click Bell	0	40	75	60	100	50	10
Dump Bin Bigbin	0	0	0	0	0	0	0
Move Can Pot	5	10	25	0	25	10	30
· · · (42 tasks in total)							
Shake Bottle	60	50	25	20	20	35	0
Stack Blocks Two	5	0	0	15	0	80	100
Stack Bowls Two	5	5	5	30	0	5	30
Stamp Seal	0	10	15	0	10	5	20
Turn Switch	0	30	0	0	5	45	0
Average SR	7.62	9.52	12.86	19.76	15.71	**36.67**	43.71
Tasks with SR > 0	15	17	22	25	19	**35**	27
Average $\mathrm{SR}_{>0}$	21.33	23.53	24.55	33.20	34.74	**44.00**	68.00

out access to instance-specific simulator ground truth. The relatively small gap to this privileged reference indicates that reusable executable affordance programs can provide competitive system-level manipulation performance under substantially weaker input assumptions.

4.3 Affordance Programs for Demonstration Generation

Evaluation Setup. We evaluate whether demonstrations generated by AffordCode can improve the generalization of a downstream visuomotor policy. Following AffordGen (Zhang et al., 2026), we use the same four ManiSkill tasks and train the same Simple-DP3 policy with 1,000 demonstrations per task. At test time, the policy observes only point clouds and robot states. For the primary comparison, AffordGen and AffordCode use the same 20×50 allocation: 20 object meshes with 50 demonstrations each.

Baselines. DemoGen (Xue et al., 2025) adapts source trajectories across object configurations; CP-Gen (Lin et al., 2025) preserves keypoint–trajectory constraints under object transformations; AffordGen (Zhang et al., 2026) transfers interaction keypoints across meshes. AffordCode instead instantiates executable affordance programs across objects and converts the resulting interactions into trajectories.

Evaluation Metrics. Following AffordGen, we report the average downstream policy success on *Source* objects under varied configurations and *Unseen* held-out meshes, together with the four-task average.

Results. As shown in Table 3, policies trained on AffordCode demonstrations achieve average success rates of 89.4% on Source objects and 65.2% on Unseen objects. Under the matched 20×50 setting, AffordCode improves over AffordGen by 8.6 and 14.2 percentage points, respectively. These results show that AffordCode-generated demonstrations provide effective supervision for downstream policy learning and improve generalization to unseen object instances.

Table 3: Downstream policy success rates (%) using 1,000 demonstrations per task. All methods use the same policy training and evaluation protocol; bold denotes the best result.

Method	Scale	Teapot	Mug	Knife	Shoe	Avg.
Source						
DemoGen	1×1000	93.3	94.0	49.0	40.0	69.1
CP-Gen	1000×1	71.3	90.0	74.7	55.0	72.8
AffordGen	20×50	92.0	96.7	79.3	55.0	80.8
AffordCode	20×50	**97.1**	**98.3**	**82.1**	**80.1**	**89.4**
Unseen						
DemoGen	1×1000	13.1	40.2	22.4	21.2	24.2
CP-Gen	1000×1	16.9	50.2	42.4	26.6	34.0
AffordGen	20×50	35.3	68.3	56.5	43.8	51.0
AffordCode	20×50	**69.2**	**73.4**	**57.1**	**61.0**	**65.2**

4.4 Affordance Guidance for a Frozen Policy

Evaluation Setup. We evaluate whether AffordCode can provide effective affordance guidance to an existing manipulation policy. Following the AffordDP setting (Wu et al., 2025), the downstream affordance interface consists of a 3D contact point and a post-contact end-effector trajectory. We consider two articulated manipulation tasks, *PullDrawer* and *OpenDoor*, and keep the downstream AffordDP policy frozen throughout the evaluation. For each task, we evaluate 100 episodes on each of three object splits: *Seen Instance*, *Unseen Instance*, and *Unseen Category*. All affordance-guided variants share the same policy checkpoint, object splits, initial conditions, execution settings, and success criteria, such that only the source and construction of affordance guidance vary.

Baselines. *Original-AffordDP* uses the original retrieval-and-transfer mechanism (Wu et al., 2025). *AffordCode-Raw* uses the initial program without verification; *AffordCode-Direct* directly synthesizes an object-level program while retaining the same verification procedure; and *AffordCode-Verified* uses the full bottom-up synthesis and verification pipeline. DP and DP3 are included as policy-level references (Chi et al., 2025; Ze et al., 2024).

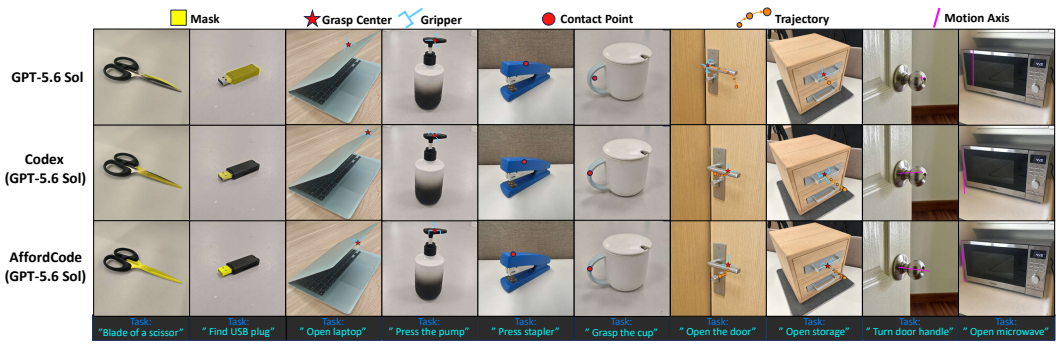


Figure 4: **Qualitative real-world affordance generalization.** Results on ten real-world object–task pairs with diverse affordance outputs under a shared GPT-5.6 Sol backbone and input setting.

Evaluation Metrics. We report success rate on *Seen Instance* (training objects), *Unseen Instance* (new objects from seen categories), and *Unseen Category* (new categories). Each task–split combination contains 100 evaluation episodes. Comparing AffordCode-Raw with AffordCode-Verified evaluates the effect of program verification and refinement, while comparing AffordCode-Direct with AffordCode-Verified evaluates the benefit of constructing object-level affordances from reusable part-level functions.

Table 4: Success rates (%) with different affordance guidance sources. Affordance-guided variants share the same frozen downstream policy; DP and DP3 are policy-level references.

Method	Seen Instance	Unseen Instance	Unseen Category
PullDrawer			
DP	17.0	7.0	2.0
DP3	37.0	8.0	3.0
Original-AffordDP	83.0	48.0	65.0
AffordCode-Raw	86.0	51.0	62.0
AffordCode-Direct	87.0	53.0	66.0
AffordCode-Verified	**92.0**	**60.0**	**72.0**
OpenDoor			
DP	20.0	15.0	3.0
DP3	37.0	5.0	5.0
Original-AffordDP	70.0	45.0	23.0
AffordCode-Raw	72.0	47.0	24.0
AffordCode-Direct	74.0	49.0	27.0
AffordCode-Verified	**80.0**	**55.0**	**33.0**

Results. As shown in Table 4, AffordCode-Verified achieves the highest success rate across all task–split combinations. Compared with AffordCode-Raw, verification and refinement improve success by 6–10 percentage points on PullDrawer and 8–9 points on OpenDoor. AffordCode-Verified also consistently outperforms AffordCode-Direct, with gains of 5–7 points on PullDrawer and 6 points across three OpenDoor splits. These results demonstrate that both program verification and the bottom-up construction of object-level affordances from reusable part-level functions contribute to more effective affordance guidance under a fixed downstream policy.

4.5 Qualitative Real-World Affordance Generalization

We qualitatively evaluate ten real-world object–task pairs. GPT-5.6 Sol, Codex, and AffordCode share the same instantiated Digital-Gene representations and task prompts. As shown in Figure 4, AffordCode more consistently produces task-aligned and spatially grounded outputs across diverse affordance forms. Full protocol details are provided in Appendix A.7. We further demonstrate real-robot execution on a scissors-to-cabinet task, with details provided in Appendix A.8.

5 Conclusion

We presented AffordCode, an agentic framework for learning reusable affordance programs over parameterized object templates. By constructing object-level programs from part-level functions and refining them through verification feedback, AffordCode supports targeted correction, compositional reuse, and generalization across compatible objects, as demonstrated across manipulation and downstream tasks.

AI USE STATEMENT

Generative AI tools were used for language editing and improving the readability of the manuscript. All AI-assisted text was reviewed and revised by the authors. The authors take full responsibility for the final content of this work.

ETHICS STATEMENT

AffordCode uses large language models to synthesize and refine executable affordance programs for robotic interaction. Model errors or hallucinations may produce incorrect interaction logic or trajectories, which could cause physical execution to deviate from the intended task and potentially lead to property damage or equipment failure. Real-robot experiments are therefore conducted in controlled environments with appropriate safety precautions.

REPRODUCIBILITY STATEMENT

To support reproducibility, we report the key implementation details, evaluation protocols, data splits, affordance acquisition settings, and model configurations used in our experiments. The Appendix provides additional details on program synthesis, verification and refinement, object parameterization, evaluation settings, prompt examples, and complete task-level results. We also provide representative Digital-Gene templates and executable affordance-program examples in the supplementary material, together with videos of the real-robot experiments.

REFERENCES

Junting Chen, Yao Mu, Qiaojun Yu, Tianming Wei, Silang Wu, Zhecheng Yuan, Zhixuan Liang, Chao Yang, Kaipeng Zhang, Wenqi Shao, Yu Qiao, Huazhe Xu, Mingyu Ding, and Ping Luo. Roboscript: Code generation for free-form manipulation tasks across real and simulation, 2024. URL https://arxiv.org/abs/2402.14623.

Tianxing Chen, Zanxin Chen, Baijun Chen, Zijian Cai, Yibin Liu, Qiwei Liang, Zixuan Li, Xianliang Lin, Yiheng Ge, Zhenyu Gu, et al. Robotwin 2.0: A scalable data generator and benchmark with strong domain randomization for robust bimanual robotic manipulation. *arXiv preprint arXiv:2506.18088*, 2025.

Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. *The International Journal of Robotics Research*, 44(10-11):1684–1704, 2025. doi: 10.1177/02783649241273668. URL https://doi.org/10.1177/02783649241273668.

Ben Eisner, Harry Zhang, and David Held. FlowBot3D: Learning 3D Articulation Flow to Manipulate Articulated Objects. In *Proceedings of Robotics: Science and Systems*, New York City, NY, USA, June 2022. doi: 10.15607/RSS.2022.XVIII.018.

Kuan Fang, Fangchen Liu, Pieter Abbeel, and Sergey Levine. Moka: Open-world robotic manipulation through mark-based visual prompting. *Robotics: Science and Systems (RSS)*, 2024.

Haoran Geng, Helin Xu, Chengyang Zhao, Chao Xu, Li Yi, Siyuan Huang, and He Wang. Gapartnet: Cross-category domain-generalizable object perception and manipulation via generalizable and actionable parts. *arXiv preprint arXiv:2211.05272*, 2022.

Haoxu Huang, Fanqi Lin, Yingdong Hu, Shengjie Wang, and Yang Gao. Copa: General robotic manipulation through spatial constraints of parts with foundation models, 2024a. URL https://arxiv.org/abs/2403.08248.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models. In *Conference on Robot Learning*, volume 229, pp. 540–562, 2023. URL https://mlanthology.org/corl/2023/huang2023corl-voxposer/.

Wenlong Huang, Chen Wang, Yunzhu Li, Ruohan Zhang, and Li Fei-Fei. Rekep: Spatio-temporal reasoning of relational keypoint constraints for robotic manipulation. *arXiv preprint arXiv:2409.01652*, 2024b.

Long Le, Jason Xie, William Liang, Hung-Ju Wang, Yue Yang, Yecheng Jason Ma, Kyle Vedder, Arjun Krishna, Dinesh Jayaraman, and ERIC EATON. Articulate-anything: Automatic modeling of articulated objects via a vision-language foundation model. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 17578–17602, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/2c8047bf3ed8ef6905351608d641f02f-Paper-Conference.pdf.

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, 2023. URL https://arxiv.org/abs/2209.07753.

Kevin Lin, Varun Ragunath, Andrew McAlinden, Aaditya Prasad, Jimmy Wu, Yuke Zhu, and Jeannette Bohg. Constraint-preserving data generation for visuomotor policy generalization. In *9th Annual Conference on Robot Learning*, 2025. URL https://openreview.net/forum?id=KSKzA1mwKs.

Zhao Mandi, Yijia Weng, Dominik Bauer, and Shuran Song. Real2code: Reconstruct articulated objects via code generation, 2024. URL https://arxiv.org/abs/2406.08474.

Kaichun Mo, Leonidas Guibas, Mustafa Mukadam, Abhinav Gupta, and Shubham Tulsiani. Where2act: From pixels to actions for articulated 3d objects. In *International Conference on Computer Vision (ICCV)*, 2021.

Danila Rukhovich, Elona Dupont, Dimitrios Mallis, Kseniya Cherenkova, Anis Kacem, and Djamila Aouada. Cad-recode: Reverse engineering cad code from point clouds. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 9801–9811, October 2025.

Erol Sahin, Maya Cakmak, Mehmet Remzi Dogar, Emre Ugur, and Göktürk Üçoluk. Afford or not to afford : A new formalization of affordances toward affordance-based robot. 2007. URL https://api.semanticscholar.org/CorpusID:31981561.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Jianhua Sun and Cewu Lu. Digital gene: Learning about the physical world through analytic concepts, 2025. URL https://arxiv.org/abs/2504.04170.

Jianhua Sun, Yuxuan Li, Longfei Xu, Nange Wang, Jiude Wei, Yining Zhang, and Cewu Lu. ConceptFactory: Facilitate 3D Object Knowledge Annotation with Object Conceptualization. In *Neural Information Processing Systems*, 2024. doi: 10.52202/079017-2402. URL https://mlanthology.org/neurips/2024/sun2024neurips-conceptfactory/.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023. URL https://openreview.net/forum?id=P8E4Br72j3.

Ruihai Wu, Yan Zhao, Kaichun Mo, Zizheng Guo, Yian Wang, Tianhao Wu, Qingnan Fan, Xuelin Chen, Leonidas Guibas, and Hao Dong. VAT-mart: Learning visual action trajectory proposals for manipulating 3d ARTiculated objects. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=iEx3PiooLy.

Shijie Wu, Yihang Zhu, Yunao Huang, Kaizhen Zhu, Jiayuan Gu, Jingyi Yu, Ye Shi, and Jingya Wang. Afforddp: Generalizable diffusion policy with transferable affordance. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6971–6980, June 2025.

Zhengrong Xue, Shuying Deng, Zhenyang Chen, Yixuan Wang, Zhecheng Yuan, and Huazhe Xu. Demogen: Synthetic demonstration generation for data-efficient visuomotor policy learning. *arXiv preprint arXiv:2502.16932*, 2025.

Yanjie Ze, Gu Zhang, Kangning Zhang, Chenyuan Hu, Muhan Wang, and Huazhe Xu. 3d diffusion policy: Generalizable visuomotor policy learning via simple 3d representations. In *Proceedings of Robotics: Science and Systems (RSS)*, 2024.

Jiawei Zhang, Kaizhe Hu, Yingqian Huang, Yuanchen Ju, Zhengrong Xue, and Huazhe Xu. Affordgen: Generating diverse demonstrations for generalizable object manipulation with affordance correspondence. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 15966–15975, 2026.

A Appendix

A.1 Scope of the Framework

AffordCode is an agentic framework for affordance learning, rather than a method for training a new foundation model. Existing language and vision-language models support program synthesis, visual inspection, and code revision. The learned artifacts are executable affordance programs and their validated updates. Our contribution concerns how object structure guides program construction and how verification feedback improves the resulting interaction logic. The object parameterization frontend supplies inputs to this process; developing a new perception backbone is outside the scope of this work.

A.2 Affordance Representation and Output Specifications

This section specifies the object interface, output conventions, and compatibility conditions used by the executable representation in Section 3.1.

Programmatic object interface. An object representation exposes its parts, geometric parameters, poses, and compositional relationships. For an object O with M parts, we write

$$P_i = \mathcal{C}_{\tau_i}(\boldsymbol{\theta}_i, T_i), \qquad \mathcal{G}_O = \left(\{P_i\}_{i=1}^{M}, \mathcal{H}_O\right), \tag{7}$$

where $\mathcal{C}_{\tau_i}$ is the template of part i, $\boldsymbol{\theta}_i$ contains its geometric parameters, $T_i \in \mathrm{SE}(3)$ is its pose in the object frame, and $\mathcal{H}_O$ records part composition. AffordCode uses this interface to query instantiated geometry and associate functional entities. The experiments instantiate this interface using ConceptFactory assets and Digital-Gene representations. A programmatic object model supplies the structural basis for affordance acquisition; it does not, by itself, specify the interaction rules required by every task. Task-relevant properties that are not provided by the representation must be derived and checked during program acquisition.

Task specification. The task description specifies the intended function and, when needed, the required output structure, interpretation, and coordinate frame. Examples can further clarify these requirements. They are part of the task context rather than an additional mandatory input interface. The notation $\mathcal{A}_c = (\mathcal{P}_c, \mathcal{O}_c)$ in the main text refers to the complete affordance program for a category and a fixed task, including its part-level and object-level routines. A trajectory returned by this program is one possible output, rather than the program itself.

Output forms. Table 5 summarizes representative interfaces. These forms are examples, not an exhaustive list or a requirement that every program return all of them. The requested task determines which quantities are exposed. Supporting a new output may require additional code, while retaining the existing entity bindings and relevant part-level computations.

Table 5: Representative affordance outputs and their interpretations. The task determines which interface a program exposes.

Output	Specification
Interaction region	A membership rule over points on the relevant part, which can be evaluated on a surface or projected as a mask.
Contact point or pose	A position, optionally paired with an orientation, expressed in a specified coordinate frame.
Motion quantity	An axis, pivot, direction, admissible range, or sequence of poses describing the requested interaction.
Applicability condition	A predicate specifying whether the interaction applies to the current object or state.
Actionability signal	A task-defined value over candidate interactions or locations, with its interpretation specified by the output interface.

Coordinate conventions. Each routine associates its geometric quantities with a coordinate frame. If $T_i = (R_i, \mathbf{t}_i)$ maps part coordinates to object coordinates, a point $\mathbf{p}_i$ and a direction $\mathbf{a}_i$ transform as

$$\mathbf{p}_O = R_i \mathbf{p}_i + \mathbf{t}_i, \qquad \mathbf{a}_O = R_i \mathbf{a}_i. \tag{8}$$

An orientation transforms by left multiplication with R_i. Translation is therefore applied to points and pivots, but not to axis directions. Object-level routines combine quantities only after resolving their frames. This distinction is particularly relevant when a contact pose is computed from a handle while a motion axis is computed from the associated door or drawer.

Compatible instances. An instance is compatible with a stored program when it supplies the required functional parts, structural relationships, and routine interfaces, and satisfies the program's applicability conditions. Dimensions, poses, and other template parameters may vary within this scope. Compatibility is thus determined by the program's requirements, not solely by an object category name. Evaluating the program with new parameters recomputes the outputs. Missing parts, unsupported structural relationships, or unmet assumptions instead require further acquisition or an explicit failure to instantiate.

A geometry-only template example. Listing 1 presents a complete box-body and cover example. The part constructors preserve the original geometric construction; mesh assembly is factored into a shared helper and a small object wrapper groups the parts. The imported `Cuboid` and `Rectangular_Ring` classes are geometric primitives from the same library. Parameters specify dimensions, wall thicknesses, offsets, and poses. Constructor rotations are in degrees and are converted to radians internally. Length parameters use the asset's consistent coordinate units. Calling `Box(body_params, cover_params)` produces the parts and their combined mesh. No contact selection, grasp pose, or interaction trajectory is defined in this example: these belong to the affordance program.

```
import numpy as np
from base_template import ConceptTemplate
from geometry_template import Cuboid, Rectangular_Ring
from utils import apply_transformation


def assemble_mesh(parts, position=(0, 0, 0), rotation=(0, 0, 0)):
    vertices, faces, offset = [], [], 0

    for part in parts:
        vertices.append(part.vertices)
        faces.append(part.faces + offset)
        offset += len(part.vertices)

    vertices = apply_transformation(
        np.concatenate(vertices), position, rotation
    )
    return vertices, np.concatenate(faces)


class Cuboidal_Body(ConceptTemplate):
    def __init__(
        self,
        top_size,
        bottom_size,
        height,
        top_bottom_offset,
        thickness,
        position=(0, 0, 0),
        rotation=(0, 0, 0),
    ):
        rotation = np.deg2rad(rotation)
        super().__init__(position, rotation)

        self.top_size = top_size
```

```
        self.bottom_size = bottom_size
        self.height = height
        self.thickness = thickness
        self.top_bottom_offset = top_bottom_offset

        h = height[0]
        tx, tb, tz = thickness

        fraction = tb / h
        middle = (
            fraction * np.asarray(top_size)
            + (1 - fraction) * np.asarray(bottom_size)
        )
        shift = fraction * np.asarray(top_bottom_offset)

        walls = Rectangular_Ring(
            h - tb,
            top_size[0],
            top_size[1],
            top_size[0] - 2 * tx,
            top_size[1] - 2 * tz,
            [0, 0],
            middle[0],
            middle[1],
            middle[0] - 2 * tx,
            middle[1] - 2 * tz,
            top_bottom_offset=np.asarray(top_bottom_offset) - shift,
            position=[shift[0], tb / 2, shift[1]],
        )

        base = Cuboid(
            tb,
            middle[0],
            middle[1],
            bottom_size[0],
            bottom_size[1],
            top_offset=shift,
            position=[0, -(h - tb) / 2, 0],
        )

        self.vertices, self.faces = assemble_mesh(
            [walls, base], position, rotation
        )


class Regular_Cover(ConceptTemplate):
    def __init__(
        self,
        outer_size,
        inner_size,
        position=(0, 0, 0),
        rotation=(0, 0, 0),
    ):
        rotation = np.deg2rad(rotation)
        super().__init__(position, rotation)

        self.outer_size = outer_size
        self.inner_size = inner_size

        w, h, d = outer_size
        iw, ih, id_ = inner_size

        top = Cuboid(
            h - ih,
            w,
```

```
            d,
            position=[0, (h + ih) / 2, d / 2],
        )

        rim = Rectangular_Ring(
            ih,
            w,
            d,
            iw,
            id_,
            position=[0, ih / 2, d / 2],
        )

        self.vertices, self.faces = assemble_mesh(
            [top, rim], position, rotation
        )


class Box:
    def __init__(self, body_params, cover_params):
        self.body = Cuboidal_Body(**body_params)
        self.cover = Regular_Cover(**cover_params)

        self.parts = {
            "body": self.body,
            "cover": self.cover,
        }

        self.vertices, self.faces = assemble_mesh(
            [self.body, self.cover]
        )
```

Listing 1: A geometry-only Digital-Gene template example. Shared primitives generate meshes; the object wrapper exposes the constituent parts without defining affordances.

A.3 Program Synthesis Details

Retrieval and composition. Retrieval considers the task, object category, and part-template composition. A compatible object-level program is retrieved with its dependencies; otherwise, relevant part functions and related object programs form the synthesis context $\mathcal{K}$. The agent checks their interfaces and applicability, binds them to the participating parts, and implements missing computations. Object-level composition preserves which parts and intermediate quantities each function depends on. Code inspection, object instantiation, geometric queries, and candidate execution support this process.

An explicit affordance program. The code example below is adapted from the microwave implementation's cuboidal handle, side-hinged door, and door-opening trajectory routines. It makes the local contact position, hinge geometry, coordinate transforms, and trajectory computation explicit. The formulas are specific to these part templates, whose stored rotations are in radians; the handle and door use the repository's respective `ZXY` and `XYZ` rotation conventions.

The `hinge_side` argument expresses a program hypothesis to be checked against the object's structure; it does not denote an extra simulator-ground-truth input. The local hinge formula applies to side-opening doors and is not a universal rule for all door types. Likewise, the contact frame is a candidate based on the cuboidal-handle convention. The resulting motion must still pass the geometric and interaction checks described below.

Output-specific extension. A region query exposes a membership rule over the relevant surface; a pose query also specifies an orientation and coordinate frame. A motion query additionally assembles an ordered sequence from the appropriate axis, range, and contact quantities. Existing bindings and computations can be retained when adding the missing output logic. Such extension may re-

quire synthesis and verification, whereas evaluating an existing program on a compatible instance does not.

Listing 2 gives an explicit example of the resulting executable interaction logic.

```
import numpy as np
from utils import apply_transformation, get_rodrigues_matrix


def part_rotation(part, order):
    # Each transformed basis vector becomes a matrix column.
    return apply_transformation(
        np.eye(3),
        [0, 0, 0],
        part.rotation,
        rotation_order=order,
    ).T


def contact_frame(handle):
    R = part_rotation(handle, "ZXY")
    local_contact = np.array(
        [0, 0, handle.size[2] / 2]
    )
    p = R @ local_contact + np.asarray(handle.position)

    return p, R


def side_hinge(door, hinge_side):
    if hinge_side not in ("left", "right"):
        raise ValueError(
            "This routine requires a side hinge"
        )

    side = -1 if hinge_side == "left" else 1

    local_pivot = np.array(
        [side * door.size[0] / 2, 0, 0]
    )

    R = part_rotation(door, "XYZ")
    pivot = (
        R @ local_pivot
        + np.asarray(door.position)
    )

    axis = R @ np.array([0, 1, 0])

    return (
        pivot,
        axis / np.linalg.norm(axis),
        side,
    )


def open_door(
    obj,
    hinge_side,
    angle_deg=50,
    steps=15,
):
    if obj.door is None or obj.handle is None:
        raise ValueError(
            "Door and handle are required"
```

```
        )

    if steps < 1:
        raise ValueError(
            "At least one motion step is required"
        )

    p0, R0 = contact_frame(obj.handle)

    pivot, axis, sign = side_hinge(
        obj.door,
        hinge_side,
    )

    trajectory = []

    for angle in np.linspace(
        0,
        np.deg2rad(angle_deg),
        steps + 1,
    ):
        motion_R = get_rodrigues_matrix(
            axis,
            sign * angle,
        )

        p = pivot + motion_R @ (p0 - pivot)
        R = motion_R @ R0

        trajectory.append((p, R))

    return trajectory
```

Listing 2: Example affordance program for opening a microwave door.

A.4 Verification, Diagnosis, and Refinement Details

Validation instances. Candidate programs are evaluated on instances sampled from valid template parameter ranges while preserving the required part composition and structural relationships. Under the default protocol, each template–task program uses eight randomly parameterized instances and at most three verification-and-refinement rounds. These instances are separate from downstream evaluation instances. After revision, the program is re-instantiated on the sampled validation instances to recompute its outputs. Task requirements and success conditions remain fixed during refinement.

A.4.1 Program Verification

Structural validity. Structural checks establish whether the candidate can be evaluated as specified. They inspect program execution, resolution of part references, compatibility of routine arguments and return values, and the types and dimensions of the exposed outputs. Numerical checks identify undefined or non-finite values and degenerate geometric quantities, such as an axis with zero norm. Frame declarations and transformations are inspected along the dependency chain so that a quantity computed in one part frame is not silently interpreted in another frame.

Geometric feasibility. Geometric checks relate the output to the instantiated object. For region outputs, they assess whether selected points belong to the intended part and satisfy the task's geometric conditions. For contact outputs, they assess the position and orientation relative to the target geometry. Where the task requires a gripper configuration or a motion, collision and clearance checks assess compatibility with the surrounding geometry. The required checks depend on the requested output; a region query does not automatically require a robot-motion feasibility test.

Visual prompting. The object is rendered together with the candidate outputs. Regions appear as surface overlays, points as markers, poses as coordinate frames, and motion outputs as axes, paths, or sampled configurations. The rendered views and task description form the visual prompt. The resulting feedback identifies relevant entities and possible spatial or functional discrepancies and is combined with geometric evidence for diagnosis.

In the microwave example of Figure 3, a candidate program places the rotation axis along the upper door edge. The visual feedback raises a concern about this hypothesis, motivating checks of the door–body connection and opening clearance. Handle placement is a cue for this inspection, rather than sufficient evidence to establish the hinge location. The corresponding part routine can then be revised to compute a side axis, after which the program is evaluated again. The red and green code lines and axes in the figure illustrate the initial and revised hypotheses, not two assignments used simultaneously in a single execution.

Interaction consistency. For outputs prescribing state changes, the rollout in Equation 4 uses geometric or kinematic state propagation, or an available execution backend. The checks assess both the interaction constraints C_t and the intended effect Φ_t in Equation 5. For drawer opening, these include motion along the sliding direction, compliance with the admissible range, and attainment of the requested opening state. A trajectory with valid numerical values can therefore still fail because it moves in the wrong direction or stops before reaching the target state. Outputs without a prescribed state transition are evaluated using the structural and geometric conditions relevant to their requested form.

These checks establish consistency under the selected validation instances and evaluation model. Geometric or kinematic propagation does not by itself establish contact-dynamic feasibility on a robot; the downstream experiments separately assess execution performance.

A.4.2 Localized Diagnosis and Code Revision

Feedback records. A failed check produces a record containing the violated condition, the involved entity, the observed discrepancy, and relevant intermediate values. Visual observations identify candidate issues, while geometric measurements and execution outcomes provide the corresponding evidence when available. The agent inspects this record alongside the source code and follows the dependencies of the affected output to identify a responsible assumption, expression, or routine. Table 6 gives representative associations rather than a fixed mapping from symptoms to causes.

Table 6: Examples of evidence used for localized diagnosis. The implicated code is determined from the candidate's dependencies.

Observed issue	Evidence to inspect	Candidate revision
Region on the wrong part	Bound entity and region membership	Entity association or region-selection logic
Contact collision	Contact offset, orientation, and clearance	Part-level contact-frame computation
Inconsistent rotation	Pivot, axis frame, and part attachments	Axis inference or coordinate transformation
Insufficient motion	Motion extent and terminal state	Range computation or object-level motion assembly

Revision scope. An error in a local geometric computation is addressed in the corresponding part routine. Errors in selecting entities, associating parts, composing routines, or specifying the task-level interaction are addressed in the object program. A revision may involve both levels when their dependencies require coordinated changes. Unrelated validated code is preserved, and the task specification is not relaxed to make a candidate pass.

Re-evaluation and termination. After each revision, the updated program recomputes its outputs and undergoes the required checks again. A candidate is admitted to the validated memory only after passing the applicable checks. A candidate that exhausts the refinement budget without passing is

recorded as an unsuccessful attempt and is excluded from validated memory. A revision is therefore retained on the basis of re-evaluation, not solely on the agent's textual assessment that the error has been corrected.

A.5 Knowledge Consolidation and Reuse

Memory initialization and admission. The part-level memory is initialized with routines available in object templates and previously generated programs. Successful acquisition updates both memory levels. Part routines are stored with their functional roles, applicability conditions, interfaces, and validation evidence, including the instances and parameter settings on which they were checked. Object-level programs retain their supported outputs and part-level dependencies. The retained knowledge consists of the validated code, rather than only a textual description of a successful attempt.

Updating shared routines. An adjustment specific to an object-level interaction remains at that level unless validation supports its use as shared part knowledge. When a shared routine changes, affected dependent programs are re-evaluated before the revision is committed. This prevents a local correction from being propagated to other compositions without checking its consequences.

Direct instantiation. For a compatible instance, the stored program is bound to its parts and evaluated using its template parameters. Interaction regions, contact frames, motion quantities, and other requested outputs are recomputed from the new geometry. This mode requires neither code synthesis nor modification of the stored interaction logic. Its scope is limited to the program's supported tasks, interfaces, and applicability conditions.

Reuse during acquisition. For a new task, composition, or output requirement, validated routines and related object programs provide context for further synthesis and refinement. Part functions can be recomposed, while object programs can serve as examples of functional composition. This mode may require new code and verification; it should not be conflated with synthesis-free instantiation of an existing program.

Separation from evaluation. Memory updates belong to the acquisition phase. In the reported evaluation protocol, the acquired programs and code memory are frozen before downstream evaluation. Evaluation objects and task-success signals are not used to revise programs or add validated entries. Consequently, the reported generalization evaluates the acquired code rather than continued adaptation on the test episodes.

A.6 RGB-D Inputs and Evaluation on RoboTwin

Template-conditioned parameterization. For RoboTwin, we construct parameterized object representations from RGB-D observations using a template-conditioned frontend. Part descriptions derived from the template guide segmentation in the RGB image. The corresponding depth observations provide geometric evidence about visible part locations and spatial extents, which assists parameter estimation. Conditioned on the RGB image, part annotations, available geometric measurements, template definitions, and few-shot examples, GPT completes the parameter assignment for object instantiation. Parameters that are not sufficiently constrained by the observations are inferred using template structure and contextual information.

Segmentation and geometric evidence. Nano Banana Pro is prompted with the RGB image and template-derived part descriptions to produce part masks aligned with the input image. For a valid pixel (u, v) in a part mask, aligned metric depth D and camera intrinsics K determine a visible surface point,

$$\mathbf{p}_C = D(u, v)K^{-1}[u, v, 1]^\top. \tag{9}$$

Here depth denotes distance along the camera optical axis. The resulting part observations provide location and extent information in a known coordinate frame. Their interpretation depends on the visible geometry and the template: the extent of a partially observed surface is not automatically the full dimension of the part. These measurements constrain or inform parameter completion rather

than guaranteeing recovery of every geometric parameter. No test-object mesh or instance-specific simulator geometry is supplied to this frontend.

Few-shot parameter examples. Parameter-completion examples pair observations and template definitions with filled parameter assignments. They illustrate parameter names, array dimensions, coordinate conventions, and how observations relate to template fields. Their object categories are disjoint from the evaluated categories. These examples supervise the input format through context; they do not provide affordance solutions for the test objects. Table 7 illustrates this format using window, washing-machine, and refrigerator templates. Each example pairs an RGB observation and its available geometric evidence with a completed template parameter record.

Table 7: Illustrative few-shot examples for template parameter completion in the RoboTwin frontend.

Object category	Template parts	Parameter completion
Window	Frame, window panels, and handles	Part dimensions, poses, and relative offsets
Washing machine	Body, door, and control components	Body and door dimensions, component poses, and offsets
Refrigerator	Body, doors, and handles	Part dimensions and poses consistent with the template's composition

Evaluation and aggregation. The 42 tasks use the Aloha-AgileX robot under `demo_clean`. Observation-based methods receive RGB-D observations and task instructions and are evaluated on 20 matched episodes per task. The RoboTwin Expert Generator is a privileged reference with access to task and simulator information; its rates are taken from the officially reported results rather than the matched episode protocol.

For observation-based methods, let $s_{k,e}$ indicate success on episode e of task k, and let $q_k = \frac{1}{20}\sum_{e=1}^{20} s_{k,e}$. We report

$$\overline{\mathrm{SR}} = \frac{100}{42}\sum_{k=1}^{42} q_k, \tag{10}$$

$$N_{>0} = \sum_{k=1}^{42} \mathbf{1}[q_k > 0], \tag{11}$$

$$\mathrm{SR}_{>0} = \frac{100}{N_{>0}}\sum_{k:q_k>0} q_k. \tag{12}$$

The last metric is defined when $N_{>0} > 0$ and is interpreted together with task coverage. Expert rates are aggregated separately using the same task-level definitions. Complete results appear in Appendix A.9.

A.7 Real-World Qualitative Evaluation

Prepared assets and shared inputs. The evaluation uses ten photographed real-world object–task pairs with corresponding parameterized Digital-Gene assets. These assets are prepared inputs to the comparison. Each includes the template code, filled instance parameters, and the object-to-image alignment needed for visualization. All three methods receive the same template code, instance parameters, task description, and output specification. The comparison therefore evaluates affordance prediction given a shared object representation; it does not evaluate the acquisition of that representation from the photographs.

A complete task-prompt example. The following grasp-center example illustrates the prompt organization and output contract. Template code and instance parameters are provided as separately delimited inputs.

> **Task.** Determine a grasp center on the designated handle of the supplied object. Return a point on the handle's graspable segment, excluding its attachment regions. Use the instantiated geometry and the part relationships provided by the object template.
>
> **Inputs.** You are given the object-template code, its filled instance parameters, and the identity of the target handle. The task input also declares the length unit used by the asset.
>
> **Output.** Return one JSON object with the fields `status`, `frame`, `unit`, and `grasp_center`. For a valid result, set `status` to `"ok"`, set `frame` to `"object"`, copy the declared length unit into `unit`, and return `grasp_center` as three finite numbers `[x, y, z]`. If the required handle is absent or no valid point can be determined, set `status` to `"unavailable"` and `grasp_center` to `null`. Do not return image-pixel coordinates or include explanatory text outside the JSON object.

Other tasks use an output-specific contract shared across methods. For example, a surface-region task specifies the domain on which its membership values are evaluated; a pose task specifies its orientation convention; and a trajectory task specifies the frame and ordering of its waypoints. The output contract is fixed before prediction.

Compared workflows. All workflows use GPT-5.6 Sol as the backbone. *GPT-5.6 Sol* directly predicts the requested result; *Codex (GPT-5.6 Sol)* uses a general-purpose coding-agent workflow, but excludes AffordCode's hierarchical memory and prescribed synthesis and verification procedure; *AffordCode (GPT-5.6 Sol)* uses the full framework. Generated programs are evaluated with the supplied instance parameters, so all methods follow the same output contract.

Projection and visualization. Final outputs are mapped onto the photographs using the same fixed object-to-image alignment for all methods. For a point $\mathbf{p}_O$,

$$\mathbf{u} = \pi(T_{CO}\mathbf{p}_O)\,, \tag{13}$$

where T_{CO} is the object-to-camera transform and π the camera projection. Points, poses, axes, trajectories, and regions are rendered with a common procedure that does not modify the predicted quantities.

Interpretation. The qualitative evaluation assesses task alignment and spatial grounding of the predicted affordances, rather than physical execution or upstream perception accuracy.

Interpretation. The qualitative figure compares task alignment and spatial grounding across masks, grasp centers, gripper poses, contact points, trajectories, and motion axes. It assesses affordance outputs on the displayed real-world assets, rather than physical execution success or the accuracy of an upstream perception system.

A.8 Real-Robot Manipulation Demonstration

Experimental setting. Given the known templates and instantiated parameters of the cabinet and scissors, AffordCode acquires task-relevant programs that GPT-5.6 Sol composes to first close the scissors and then place them into the cabinet. The full execution video is provided in the supplementary material.

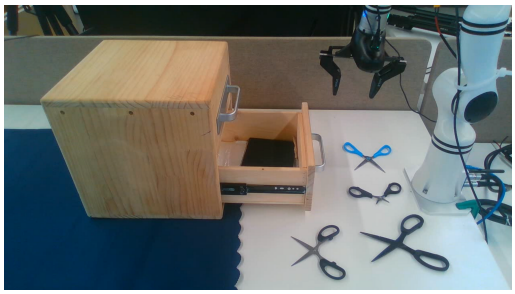


Figure 5: **Real-robot manipulation setup.** Cabinet and scissors used for sequential affordance-program execution.

A.9 Full RoboTwin Results

Table 8 reports all 42 tasks from the evaluation in Section 4.2. Observation-based methods use 20 matched episodes per task. The RoboTwin Expert Generator is listed separately as a privileged reference and uses its officially reported rates. The summary rows use the aggregation rules in Appendix A.6; the conditional average includes only tasks with nonzero success.

Table 8: Complete RoboTwin success rates (%). Bold marks the best observation-based result for each task, with all-zero rows left unmarked. The privileged expert reference is excluded from this comparison.

	Observation-Based Methods						Privileged Reference
Task	VoxPoser	CoPa	RoboScript	ReKep	MOKA	**AffordCode**	RoboTwin Expert
adjust_bottle	0	5	0	20	0	**25**	100
beat_block_hammer	0	0	0	**35**	0	10	53
blocks_ranking_rgb	5	0	0	15	0	**60**	80
blocks_ranking_size	0	0	0	0	0	**60**	80
click_alarmclock	**80**	45	70	50	35	50	0
click_bell	0	40	75	60	**100**	50	10
dump_bin_bigbin	0	0	0	0	0	0	0
hanging_mug	0	0	0	0	0	0	0
move_can_pot	5	10	**25**	0	**25**	10	30
move_pillbottle_pad	0	0	20	0	25	**35**	50
move_stapler_pad	0	0	0	0	0	**75**	100
open_laptop	0	0	5	30	5	**40**	0
open_microwave	5	15	5	10	0	**50**	0
pick_diverse_bottles	0	0	0	0	0	**15**	62
place_a2b_left	30	15	50	55	45	**60**	50
place_a2b_right	30	15	50	60	50	**65**	60
place_bread_basket	0	0	25	20	0	**45**	0
place_burger_fries	0	0	5	65	0	**75**	100
place_can_basket	0	0	**25**	0	0	0	0
place_cans_plasticbox	0	0	10	0	0	**50**	100
place_container_plate	5	10	35	**60**	35	20	82
place_dual_shoes	0	0	0	0	0	**50**	22
place_empty_cup	20	0	10	55	**65**	20	85
place_fan	0	0	0	0	0	**45**	70
place_mouse_pad	10	0	10	40	45	**80**	100
place_object_basket	0	0	**15**	10	0	0	0
place_object_scale	30	10	20	10	15	**60**	80
place_object_stand	5	10	0	**55**	35	50	90
place_phone_stand	0	0	0	**5**	0	0	0
place_shoe	0	10	0	20	30	**70**	100
press_stapler	0	70	35	70	**75**	55	0
put_bottles_dustbin	0	0	0	0	0	0	0
put_object_cabinet	0	0	0	0	0	0	0
rotate_qrcode	0	0	0	5	15	**30**	80
shake_bottle	**60**	50	25	20	20	35	0
shake_bottle_horizontally	25	**50**	5	15	25	35	0
stack_blocks_three	0	0	0	0	0	**75**	82
stack_blocks_two	5	0	0	15	0	**80**	100
stack_bowls_three	0	0	0	0	0	**5**	20
stack_bowls_two	5	5	5	**30**	0	5	30
stamp_seal	0	10	**15**	0	10	5	20
turn_switch	0	30	0	0	5	**45**	0
Average SR	7.62	9.52	12.86	19.76	15.71	**36.67**	43.71
Tasks with SR > 0	15	17	22	25	19	**35**	27
Average $SR_{>0}$	21.33	23.53	24.55	33.20	34.74	**44.00**	68.00