VIA: VISUAL INTERFACE AGENT FOR ROBOT CONTROL

Anonymous authors
Paper under double-blind review

ABSTRACT

Robot manipulation is a complex task that requires visual perception, physical reasoning, planning, and closed-loop control. General-purpose foundation models (FMs) have grown remarkably capable of some of these, especially perception and reasoning. Inspired by the growing ability of FM-powered agents to operate software through visual interfaces, we ask whether that same competence suffices to control a robot *directly* given the right interface. We present VIA (**V**isual **I**nterface **A**gent), a framework that recasts robot control as an agentic task where an off-the-shelf agent drives a manipulator directly through an interface. The interface is a virtual workspace of interactable visual components where the agent can probe pixels of interest with mouse-like tools and command a virtual end-effector to set new target poses with a small set of general tools. We show that VIA enables agents of varying capabilities to perform diverse tabletop manipulation tasks in simulation. We also use VIA to drive an off-the-shelf mobile manipulation robot in the real world to complete tasks that require both navigation and manipulation. Both settings are zero-shot, requiring no robot training or additional action primitives. Performance scales well with the size and strength of the underlying FMs. These results suggest that frontier agents already possess skills that transfer directly to robot control given the right interface: *your coding or computer-use agent is, in a sense, secretly a robot-control agent*.

1 INTRODUCTION

General robot manipulation is a long-standing research goal at the intersection of robotics and learning. It requires policies that are capable of complex visual understanding, physical reasoning, long-term planning, and precise closed-loop control. Foundation models (FMs) (Radford et al., 2019; Bommasani et al., 2021; OpenAI, 2023) trained on internet-scale data have become remarkably capable of several of these, such as visual perception and reasoning, which has made leveraging them for robotic manipulation highly attractive.

One approach for this is to convert existing vision-language models into vision-language-action (VLA) models, by fine-tuning them on robot data to produce low-level actions (Zitkovich et al., 2023; Black et al., 2025). However, due to a lack of both compute and robot-specific data, VLAs are often orders of magnitude smaller than frontier FMs. This gap has a profound impact on capabilities, particularly the physical reasoning and long-horizon planning skills that current robot policies lack.

Another paradigm is Code-as-Policies (CaP) (Liang et al., 2023; Singh et al., 2023), where FMs generate programs to control robots using perception APIs and hand-crafted skill primitives, e.g., `mask = segment_text_prompt("green cube")`, `grasp_pose = sample_grasp_pose("red cube")` (Fu et al., 2026). Although this is a natural way to tap into the strong coding capabilities of FMs, the FMs are not directly involved during rollouts. Therefore, the system is often bottlenecked by human-crafted perception and action abstractions rather than by the FM.

We propose a new way to connect FMs to robot manipulation: recasting robot control as a visual tool-use task for agents by introducing a virtual workspace that the agent can probe and interact with, similar to general computer use. Agents (Yao et al., 2023) wrap FMs with harnesses to enable better performance on long-horizon tasks (Yang et al., 2024; Jimenez et al., 2024), as well as closed-loop control of software or games in virtual environments (Anthropic, 2025a). Frontier agents can now operate real software through native screenshots, clicking, and keyboard operations (Shi et al., 2017;

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.

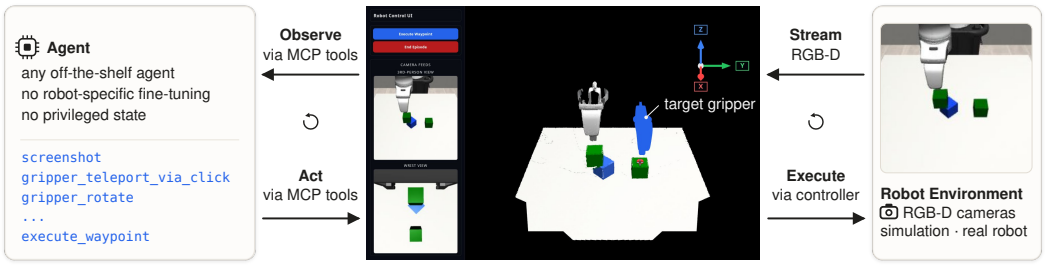

Figure 1: **VIA overview:** an off-the-shelf agent controls a robot through an *interactive* interface. **Middle:** The interface includes a target gripper (blue) rendered on top of the real scene to form a virtual workspace. In this example, the workspace contains a 3D point-cloud scene reconstructed from RGB-D cameras, as well as raw third-person and wrist camera feeds. In real-world scenarios where the point clouds are sparse and the depth measurements are noisy, we render the virtual gripper on top of raw RGB camera feeds. **Left:** The agent observes and acts through human-legible MCP tools: e.g., it can take screenshots, move the virtual gripper with `gripper_{ops}`, observe the resulting change in the interface (*without moving the actual gripper*), and refine the target through additional tool calls. **Right:** Once satisfied with the target gripper's state (the *waypoint*), the agent invokes a controller to move the real gripper to that target. The agent repeats this loop, replanning from a new observation after each tool call and completing the task through a sequence of waypoints.

Anthropic, 2024a; OpenAI, 2025b). They perform long-horizon tasks in a closed-loop manner, iteratively observing, planning, and executing, remarkably similar to what we want from robot policies. Therefore, we ask: given the right *visual interface* and human-legible tools, can a state-of-the-art agent control a robot without any robot-specific adaptation?

We present VIA (**V**isual **I**nterface **A**gent), a framework for robot control that lets off-the-shelf visual agents drive robot manipulators through an interface (Fig. 1) and a controller. The interface is a virtual workspace where the agent interacts with the scene, staging and refining controller commands without affecting the real environment. The core component is a virtual *target gripper* that the agent can move around with a small set of human-legible Model Context Protocol (MCP) tools (Anthropic, 2024b), much as a human would pose an object in 3D design software such as Blender. Through the interface, the agent reasons about the scene, issues MCP tool calls, and inspects their visual effects to refine the target gripper's state until satisfied. The agent submits the refined target gripper state to the controller as a *waypoint*, and the controller moves the real gripper to that target. The agent then observes the updated scene and plans the next waypoint, repeating until the task is complete.

We evaluate VIA with two frontier agents, Claude Code (Anthropic, 2025b) and Codex (OpenAI, 2025a), across various model tiers. Our evaluation suite includes six simulated tabletop manipulation tasks and five real-world mobile manipulation tasks, spanning pick-and-place, articulated objects, precise execution, and long-horizon multi-object assembly. VIA achieves strong results, solving these tasks end-to-end in a zero-shot setting with high success rates at an estimated API cost of a few dollars per successful episode on most tasks. We show that VIA's performance scales well with the underlying model capacity, achieving success rates of 38%, 73%, and 100% with Codex using Terra 5.6, Sol 5.6, and Astra 6, respectively, and showing a similar trend for Claude Code. VIA can also consume demonstrations written as plain text: providing a list of example waypoints increases VIA's success rate with Claude Code from 62% to 75% and with Codex from 68% to 74% on simulated tasks. VIA opens up a new opportunity for leveraging state-of-the-art FMs at scale in robotics and demonstrates that, with the right interface, robot control can become another economically valuable agentic task that benefits directly from modern models and agents. Since VIA uses frontier models and agents unmodified, improvements in agent perception, reasoning, context management, and tool use transfer directly to robot control.

2 Related Work

We review the following areas related to our work: existing ways to leverage FMs for robot manipulation, waypoint-based robot control, and computer-use agents.

2.1 Foundation Models for Robot Manipulation

Vision-Language-Action (VLA) models. VLAs fine-tune vision-language models (VLMs) on robot data to directly predict low-level actions (Zitkovich et al., 2023; Kim et al., 2024; Black et al., 2025). While these models have shown that FMs can enhance robot policies, their generalization abilities often still lag behind those of their base FMs, in part because fine-tuning on action data can degrade their reasoning and multimodal understanding abilities (Hancock et al., 2026).

FMs in Hierarchical Robot Control. Hierarchical approaches use FMs for high-level planning and reasoning while delegating low-level control to learned robot policies. SayCan (ichter et al., 2023) combines language-model planning with learned or scripted skill affordances to select which skill to execute. Text2Motion (Lin et al., 2023) extends this by verifying the geometric feasibility of the planned skill sequence with learned skill value functions. More recent approaches use VLMs to guide VLAs with subtask instructions (Shi et al., 2025; Chen et al., 2026), 3D manipulation policies with predicted paths (Li et al., 2025), or diffusion policies with latent plans (Huang et al., 2025). FMs can also guide policy learning rather than execution: de Bakker et al. (2025) use VLM-generated keypoint trajectories as scaffolds and dense rewards to train a residual RL policy in simulation, so the VLM is no longer needed at test time.

Visual Prompting for Manipulation. PIVOT (Nasiriany et al., 2024) queries a VLM to select among visually annotated action candidates sampled from an iteratively refined Gaussian proposal distribution. MOKA (Fang et al., 2024) similarly prompts a VLM to select affordance keypoints and coarse waypoint parameters from marked images, relying on object segmentation, grasp proposals, and hand-designed motion routines to generate robot motion.

Code-as-Policies. Code-as-Policies (CaP) uses FMs to write code to control a robot (Liang et al., 2023; Singh et al., 2023). The generated code, rather than the FMs themselves, is the policy that handles perception and control. As a result, the code policies often rely on perception APIs like SAM-based segmentation and CLIP-based object retrieval to parse image observations, and rely on action primitives like grasping and placement modules (Liang et al., 2023; Huang et al., 2023a;b) to handle actions. CaP-X (Fu et al., 2026) adds multi-turn execution feedback and allows the coding FMs to revise their generations after each attempt, but the policy still does not leverage the general power of the FMs during execution.

Comparison with VIA. VIA operates in a distinct setting different from the four categories listed above. Compared to VLAs, VIA uses off-the-shelf FMs and agents without any fine-tuning. In contrast to VLMs in hierarchical settings, VIA does not involve any low-level VLAs for translating instructions into robot actions. VIA allows agents to specify controller targets directly in the controller's native degrees of freedom (DoF), avoiding any action abstractions or action-proposal primitives used in visual prompting. Finally, unlike CaP, which uses the generated program as the policy, the agent in VIA *is* the policy and remains in the control loop during execution.

2.2 Waypoint-based Robot Control

Waypoint-based policies shorten the decision horizon by predicting sparse end-effector targets executed by a controller or motion planner (Shridhar et al., 2023; Goyal et al., 2023). HYDRA (Belkhale et al., 2023) and SPHINX (Sundaresan et al., 2025) combine waypoints for long-range motion with dense actions for precise manipulation. VIA builds on SPHINX's visual teleoperation interface. Instead of having a human teleoperator to collect data using this interface, VIA uses off-the-shelf FM agents to specify and refine waypoints to control the robot directly without robot-specific training.

2.3 Computer-Use Agents

A rapidly growing line of work equips FMs to operate computer software directly through visual interfaces and a small set of generic tools (e.g., taking screenshots, clicking, typing) (Anthropic, 2024a; OpenAI, 2025b). Benchmarks such as OSWorld (Xie et al., 2024) and WebArena (Zhou et al., 2024) measure this competence on real desktop and web environments, and coding agents extend it to multi-step software tasks (Jimenez et al., 2024; Anthropic, 2025b). These agents exhibit strong closed-loop visual agency: they perceive an interface, act through tools, observe the outcome,

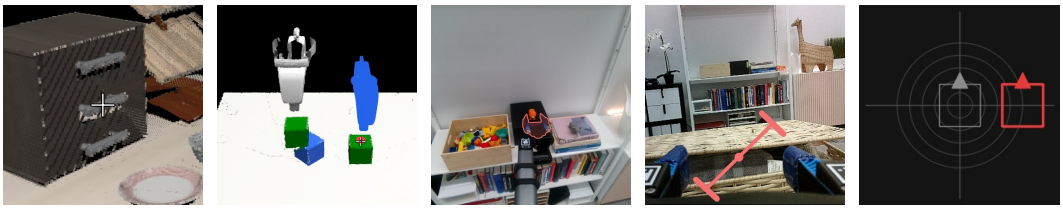

Figure 2: **Examples of interactions in VIA.** The core idea of VIAis to provide visual feedback after each tool call so that the agent can experiment with the target in this virtual workspace. (a) *hover*: draws a crosshair at snapshot coordinates (u, v) so the agent can check whether its predicted point falls on the intended target (e.g., the handle of the middle drawer). It also returns depth information when available. (b) *click*: moves the target gripper (blue) to an approach position near the red point clicked by the agent. (c) *translate*: moves the target gripper (red, contour) given 3-DoF offsets. (d) *rotate*: rotates the target gripper (red, glyph) given 3-DoF rotation deltas. (e) *nav*: shows the staged movement of the 3-DoF mobile base, when available.

and recover from mistakes. VIA repurposes this competence for robot manipulation by simply presenting the robot as another application the agent can operate, so that progress in computer-use agents transfers directly to robot control.

3 VIA: Visual Interface Agent for Robot Control

VIA recasts robot control as an agentic observe-act loop over a visual interface. An agent operates a robot-control UI much as a person would: it takes a screenshot, reasons about what it sees, issues intuitive commands through Model Context Protocol (MCP) tools that stage a virtual target gripper visually representing the input to the controller. It then observes the result, and adjusts until satisfied before invoking the controller to actually move the robot. It continues the loop until the task is complete or the episode times out. We describe the interface in Section 3.1 and the agentic loop in Section 3.2. The detailed list of MCP tools is provided in Appendix A.

3.1 Visual Interface

The interface is a collection of interactive visual components that the agent can observe from and interact with. We first explain how the agents observe the world through the interface, and then how they control the robots by interacting with it.

All interface components are constructed from raw sensor inputs and robot proprioceptive information, without access to privileged state. When the depth readings are dense and reliable, such as in tabletop settings or in simulation, we reconstruct a 3D point cloud from RGB-D inputs and use it as the main workspace. When the depth readings are sparse, such as in our real-world experiments where a mobile base operates in a room, we simply show the 2D images. The 3D point cloud allows the user (either a human or an AI agent) to change the viewport by applying the `orbit`, `pan`, or `zoom` functions to a virtual camera, enabling views beyond those of the fixed cameras used to construct the point cloud. The agent observes this point cloud through snapshots of the current viewport, which it can modify, rather than the raw vectors of the points. The interface can be presented as a graphical user interface with a defined layout (as shown in the center of Fig. 1), or passed directly to the agent as a structured list of named, typed entries, such as JPEG image bytes and proprioceptive values.

An agent controls the robot by deciding inputs to the controller. For example, when controlling a single robot arm with a gripper, the controller inputs are 6-DoF target poses and a binary open/close state; when controlling an arm with a mobile base, the controller may additionally take 3-DoF base movement parameters. We refer to this set of target values as a *waypoint*. Instead of imagining and setting a precise target value directly given RGB-D inputs, the interface in VIA acts as a virtual workspace that allows the agent to probe and experiment with the target visually.

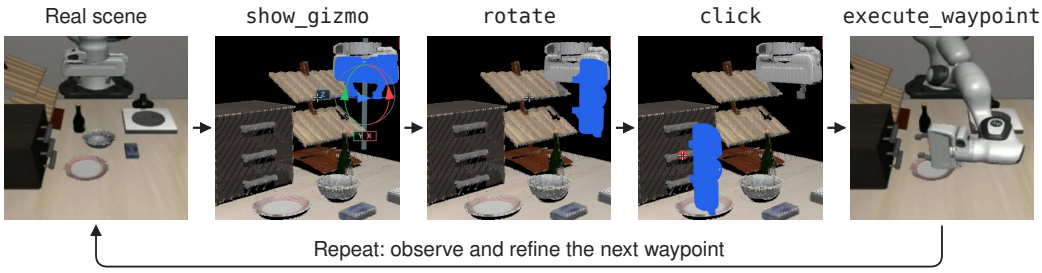

Figure 3: **One round of the VIA control loop**, shown on the *open drawer* task. From left to right, the agent observes the real scene, then refines a waypoint through `show_gizmo`, `rotate`, and `click` (tool names abbreviated). The three target previews show successive states of the same waypoint, with each tool call returning an updated snapshot. After satisfied with the waypoint, the agent calls `execute_waypoint` to pass the waypoint to the controller, which moves the real robot. Then the agent observes the outcome and plans the next waypoint, repeating until termination. Note that the viewport has been adjusted by the agent to look towards the drawer in previous tool calls.

Fig. 2 illustrates the most representative interactions that the interface supports. The core component is a virtual illustration of the *target waypoint* that visually shows the staged target transformation. For a robot arm manipulator, the target gripper is rendered as a 3D model in the point-cloud workspace or as a contour or line-and-dot glyph over a camera image. For a mobile manipulator, this may also include a visualization of the base movement. Each adjustment updates the visualization, allowing the agent to inspect and revise the target without moving the real robot. Additionally, helper tools such as `hover` render visual cues to help the agent verify its prediction or probe additional information from the environment, such as depth, when available.

3.2 Agentic Loop

An agent controlling a robot through a controller runs a simple loop: observe the scene, call the controller with a waypoint, observe the outcome, and repeat until a terminal condition is met. This closed-loop structure lets the agent recover from errors and re-plan based on what it observes, but the only feedback on a waypoint arrives after the robot has already moved, so every misjudged target costs a real motion.

With the interface, VIA keeps this outer loop but nests a *mini-loop* inside each iteration. Instead of committing a waypoint in one shot, the agent edits the target waypoint through any number of interface operations, each of which returns an updated rendering of the staged target on top of the real scene, and keeps revising until satisfied. Then the agent calls `execute_waypoint`, which invokes a controller that moves the real gripper to the target pose while obeying physical constraints. The mini-loop allows for trial-and-error to happen in virtual workspace rather than the physical robot: the agent verifies each waypoint visually before executing it, so the real robot only ever receives waypoints that have already been checked virtually. Fig. 3 shows one round of this loop, with VIA refining and then executing a single waypoint to prepare the gripper for opening a drawer.

We drive VIA using general-purpose agents, with no robot-specific fine-tuning or additional scaffolding, to demonstrate that, given the right interface and MCP tools, existing agents and models can be strong robot manipulators. VIA reuses off-the-shelf agents like Claude Code or Codex, including their tool-use loops and context management, so improvements in both the FM and the agent software can transfer to robot manipulation. We equip the agent with a short system prompt that covers the basics of the interface and provides general guidance on using it effectively, such as the configuration and limitations of the robot, the concept of waypoints and preference for closed-loop verification over open-loop execution (full prompt in Section E.1).

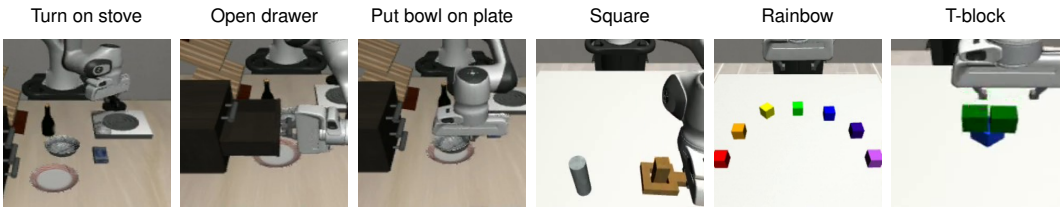

Figure 4: **The six evaluation tasks.** Each panel shows the final state of a successful VIA rollout, captured by a third-person camera. Object placements are randomized across seeds.

4 Experiments

We evaluate VIA on a suite of six tabletop manipulation tasks with a Franka Panda arm in simulation to demonstrate how VIA scales with different models, compare it with a naive baseline, and examine whether it benefits from detailed text demonstrations. Then, we run VIA on five real world manipulation tasks with a Stretch 4 mobile robot to show that the system works well in real world with different hardware and noisy perception.

4.1 Simulation Experimental Setup

Tasks. We use the following tasks in our evaluation (Fig. 4):

- *Turn on stove*: Turn on a stove by rotating its knob, from LIBERO-Goal (Liu et al., 2023).
- *Open drawer*: Open the middle drawer of a three-drawer cabinet, from LIBERO-Goal.
- *Put bowl on plate*: Pick up a bowl and place it on the center of the plate, from LIBERO-Goal. This demands reasoning on the grasp point and precision, as the bowl must be grasped by pinching its side wall and placed at the exact center of the plate to succeed.
- *Square*: Pick up the square nut and fit it onto the square peg, from robosuite (Zhu et al., 2020).
- *Rainbow*: Arrange seven randomly placed colored blocks into a rainbow, a task we created. This tests long-horizon planning and execution.
- *T-block*: Build a T-shape with three blocks of the same size: one blue block on the bottom and two green blocks resting directly on top of it, side by side; from BuilderBench (Ghugare et al., 2025), re-implemented. This requires both solving a physics puzzle and precise manipulation.

These tasks test a diverse set of skills: manipulation, physical reasoning, long-term planning, and precise execution. All tasks run in the robosuite simulator. We use a simple PI (proportional-integral) controller that linearly interpolates between the current state and the target waypoint. The outcome of each task is a binary success or failure.

Evaluation Protocol and Baseline. We use Claude Code (CC) and Codex as our agents. To evaluate how VIA scales with the size and strength of the underlying model, we run CC with Sonnet 5, Opus 4.8[1], and Fable 5, and run Codex with Terra 5.6, Sol 5.6, and Astra 6. We run all agents at `xhigh` reasoning effort. On top of the shared system prompt in Section E.1, we give the model a simple task prompt by default, stating only the task description and success condition but no detailed instructions. We evaluate each agent-model configuration over 10 seeds per task, with varying initial object placements. We cap each episode at one hour of wall-clock time, although the agents complete most tasks in far less time when successful. We disable memory access for the agents so that they cannot accumulate experience across episodes. We also disable internet search and file access to prevent reading the exact task state specifications from the codebase[2].

To isolate the benefit of VIA, we consider a naive *baseline* that uses the same inputs, agent, and controller setup as VIA but does not have access to the interface. It receives the same observations (snapshots of the same images and 3D point clouds, along with proprioceptive states) but must issue

[1]Opus 4.8 was the strongest model for a considerable amount of time during which this project was developed, so we decided to use this model instead of the newer Opus 5 to avoid the cost of re-evaluating.

[2]We find that agents may cheat by trying to read the source files of the simulation benchmarks to recover the exact simulation states and object positions if we do not disable general file access.

Task	Claude Code			Codex		
	Medium Sonnet 5	Large Opus 4.8	Extra Fable 5	Medium Terra 5.6	Large Sol 5.6	Extra Astra 6
Turn on stove	70%	100%	100%	70%	100%	100%
Open drawer	40%	70%	90%	30%	80%	100%
Put bowl on plate	50%	60%	100%	70%	100%	100%
Square	30%	70%	80%	0%	30%	100%
Rainbow	70%	80%	100%	50%	100%	100%
T-block	0%	10%	40%	10%	30%	100%
Overall	43.3%	65.0%	85.0%	38.3%	73.3%	100.0%

Table 1: **Performance of VIA in simulation.** Success rates over ten seeds per task.

commands to the controller for immediate execution, without the ability to probe and iterate using any interface components.

VIA unlocks additional opportunities to improve its performance without any gradient-based fine-tuning. Similar to other agentic systems powered by FMs, we can optimize its prompts or ask the agents to build new tools to improve task performance. We perform a preliminary case study along this line by giving agents detailed task prompts explaining how we would solve certain tasks with example lists of waypoints. We refer to this variant as *VIA-detailed*.

4.2 Simulation Experimental Results

Fig. 5 shows the scaling curve of VIA with different model tiers, measured by average success over all six tasks, as well as a comparison with the baseline. Table 1 shows the per-task success rates for VIA, and Table 7 in Section C shows them for the baseline. Table 2 shows how having detailed instructions in prompts affects performance for different agents. We report tool call efficiency and estimated API costs in Section B.

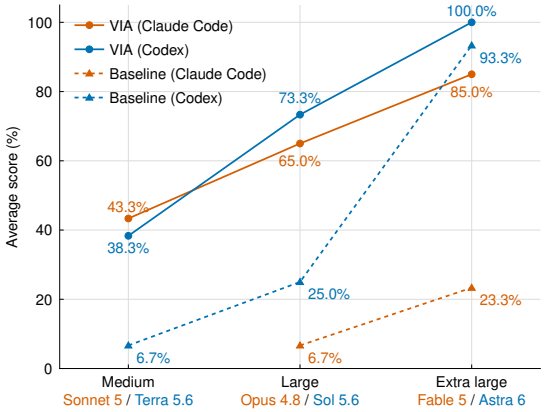

Figure 5: **Performance of VIA and the baseline in simulation.** VIA scales with the capacity of the underlying models and achieves a perfect score using Astra. The baseline performs poorly except when using Astra.

VIA solves diverse manipulation tasks zero-shot. VIA achieves strong performance zero-shot in our simulation suite, achieving around 40% success even with the smaller models (Sonnet, Terra). The high success rates on *Turn on stove* and *Open drawer* indicate that VIA is able to handle articulated objects, beyond simple pick-and-place motions. The *Rainbow* task is specifically designed to test long-horizon planning and execution. We observe that the agents always plan carefully before moving any block, estimating each block's location and choosing the ordering with the fewest moves. The effect of such planning is visible in their behavior: they sometimes assemble the rainbow with red on the left and sometimes with red on the right to minimize the number of motions required depending on the initial distribution of the blocks. *T-block* remains challenging for all except for Codex with Astra. All agents figure out the right strategy is to rotate the bottom blue block by 45 degrees to increase the contact area, but most of them fail to place the green blocks precisely on each half of the blue block, as this task tolerates only about 8 mm of placement error.

VIA scales with the capacity of the underlying models. Fig. 5 shows that VIA benefits directly with the scale of the underlying models. Performance of VIA improves from 43% to 85% by switching to bigger models in Claude Code and from 38% to a perfect 100% for Codex.

	Claude Code Opus 4.8		Codex Sol 5.6	
Task	Default	Detailed	Default	Detailed
Turn on stove	**100%**	**100%**	**100%**	80%
Open drawer	70%	**100%**	**80%**	70%
Put bowl on plate	60%	**100%**	**100%**	**100%**
Square	**70%**	60%	30%	**80%**
T-block	10%	**20%**	30%	**40%**
Avg.	62%	**76%**	68%	**74%**

Table 2: **Default versus detailed prompts.** Success rates of VIA under default and detailed prompt variants, each evaluated over 10 episodes per task. Bold marks the higher success rate, including ties. Detailed prompts bring moderate improvements by supplying the agents with lists of example waypoints written by humans.



Figure 6: **The five real-world evaluation tasks.** Red arrows indicate the intended motions. Initial locations of the objects and the mobile Stretch 4 robot are both randomized for each episode.

Method	Lamp	Drawer	Towel	Wipe	Basket
VIA	4/5	5/5	5/5	5/5	5/5

Table 3: **Real-world evaluation.** Success rates of VIA (w/ Codex Astra 6) on real-world tasks.

VIA greatly outperforms the baseline. Compared to the naive baseline (dashed lines in Fig. 5), which uses the same observations and controller but does not use the interface, VIA performs substantially better across all configurations except for Astra 6, where the interface improves performance only slightly, from 93% to 100%. The surprisingly strong performance of the baseline with Astra 6 is also consistent with many recent demonstrations showing that Astra 6 can perform many robot manipulation tasks zero-shot. Our evaluation quantifies this step change by comparing performance across models, highlighting how poorly prior state-of-the-art models perform without VIA.

Detailed text instructions bring modest improvement. In Table 2, we take tasks where at least one VIA agent in the *large* configurations from Fig. 5 scores less than 100%. We manually write detailed instructions for every task explaining how we would complete the task using this interface and feed the instructions to the agents as additional prompts. Overall, the instructions help modestly, improving the success rate by 14% for CC Opus and 6% for Codex Sol. The remaining failure modes vary from task to task. For example, T-block failures are primarily due to imprecise perception and execution; Codex Sol's failures in *drawer* are caused by opening the wrong drawer, which is not specifically addressed in the instructions. A more promising direction may be to have the agents review their own failures and optimize prompts accordingly instead of having a human write them.

4.3 Real-World Evaluation

For the real robot experiments, instead of staying in the stationary tabletop domain, we use a mobile manipulator, Stretch 4 (illustraions and details in Section D), that operates at room scale to demonstrate that the system works with coarse and noisy depth perception. The mobile robot also allows for a much wider range of random initial conditions, as well as multi-step tasks that require the agent to remember objects that go out of sight as it moves the robot. Unlike in simulation, where we have perfect depth information to construct dense, informative point clouds, the Stretch

4 robot has a head-mounted lidar sensor that provides depth information only for sparse points at room scale. Therefore, we use a collection of purely 2D-based UI components for the interface (illustrations in Fig. 2 (c) and (d)) but allow the agents to probe depth with `hover`. We run Fast-FoundationStereo (Wen et al., 2026) for the stereo camera on the wrist to provide better depth estimation in the wrist view.

Tasks. We use the following tasks in our real-world evaluation (Fig. 6). Apart from the task-specific randomization described below, the initial location of the robot is also randomized.

- *Lamp*: Turn on the lamp by pulling the small chain downward. The lamp may not appear in the initial view of the robot. The chain is small and requires precise motion.
- *Shelf*: Open the top drawer and put toy lion from the shelf into the drawer. The toy lion's location and pose are randomized over the shelf. The toy and drawer are far apart so that they do not appear in the robot's view at the same time.
- *Towel*: Fold a towel diagonally by bringing one corner to the opposite corner. The location and pose of both the towel and the table are randomized. This tests deformable object manipulation.
- *Wipe*: Pull a wipe out of its container and use it to clean the black mark on the table. The location and shape of the black mark, as well as the location and orientation of the table, are randomized.
- *Basket*: Open the basket until the lid flips to the back of the basket. The location and orientation of the basket are randomized. This manipulation of an articulated object requires complex motion.

Results. Table 3 summarizes the results. We evaluate VIA with Codex Astra 6 as the agent for five episodes in each task, and it achieves a near-perfect $24/25$ (96%) success rate. These results show that VIA extends to real-world manipulation with commercial hardware. The *Towel* and *Wipe* tasks extend the evaluation suite to deformable objects. Apart from the quantitative results, these evaluations also exhibit several interesting behaviors. In *Lamp*, the task description mentions that the lamp is on a cabinet, and the agent is able to search for the lamp quickly once the cabinet appears in its view. The one failure in this task is when the agent overextends the gripper to hold the stem of the lamp instead of the chain, which could be a perception problem as the chain and the stem are the same silver color. In *Shelf*, the head camera's limited field of view requires the agent to remember the locations of the lion and drawer to move efficiently between them. VIA does this using the underlying agent's built-in context management, without any additional engineering. In *Towel*, the agent is able to fold the towel precisely so that the edges align well. In *Wipe*, the agent's view is blocked when wiping, and it automatically moves the gripper away after a few wipes to check the cleanliness of the table. In one episode, it drops the wipe on the table and pushes it with a closed gripper instead of holding it, after the latter strategy fails to wipe cleanly. Finally, in *Basket*, the agents automatically retries within episodes when the lid slips out of the gripper during opening.

5 Conclusion

We present VIA, a framework that recasts robot control as a visual agentic task: an off-the-shelf FM-powered agent operates an interface built from RGB-D streams, posing target grippers and executing waypoints through human-legible MCP tools. Using off-the-shelf frontier agents, VIA solves six simulated manipulation tasks and succeeds in $24/25$ trials across five real-world mobile manipulation tasks, without robot-specific fine-tuning or privileged state. Performance improves with model strength: each new generation of general capability transfers to the robot, with no fine-tuning required. One direction for future work is to include a more sophisticated controller, which could allow the system to avoid collisions or handle dynamic tasks. Another direction is to study how to extend similar ideas to dexterous manipulation beyond parallel grippers. Even in its current form, however, VIA has practical uses: assembly tasks where reliability matters more than speed, reset mechanisms for real-world reinforcement learning, and data collection in which a slow interface agent generates demonstrations for training fast policies.

Limitations. VIA achieves high success rates with the frontier agents, but inference remains slow and expensive. We expect similar performance from smaller and cheaper models as FMs continue to improve, with open-weight or less congested models further reducing cost and latency. Slow inference also restricts VIA to quasi-static tasks: it is not suitable for dynamic tasks like catching a ball. However, a wide range of quasi-static tasks carry significant value and existing methods are not yet reliable on them, making VIA a compelling option in this regime.

AI USE STATEMENT

In this work, we used generative AI tools for assistance with code implementation, running experiments, creating figures, and polishing the manuscript. The core idea, the method, and the experiments are all conceived and designed by the authors. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work tackles applications of foundation models and agents in robotic control. We are not aware of any ethical issues regarding this work.

REPRODUCIBILITY STATEMENT

We have provided details to reproduce our experiments, including documentation of our MCP tools and prompts, in Appendix A and Appendix E. We also plan to fully open source the code and video recordings of the evaluation episodes.

REFERENCES

Anthropic. Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku. https://www.anthropic.com/news/3-5-models-and-computer-use, 2024a.

Anthropic. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol, 2024b.

Anthropic. Claude 3.7 Sonnet and Claude Code. https://www.anthropic.com/news/claude-3-7-sonnet, 2025a.

Anthropic. Claude Code. https://claude.com/product/claude-code, 2025b.

Suneel Belkhale, Yuchen Cui, and Dorsa Sadigh. HYDRA: Hybrid Robot Actions for Imitation Learning. In *Conference on Robot Learning (CoRL)*, pp. 2113–2133, 2023. URL https://proceedings.mlr.press/v229/belkhale23a.html.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Robert Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Lucy Xiaoyang Shi, Laura Smith, James Tanner, Quan Vuong, Anna Walling, Haohuan Wang, and Ury Zhilinsky. π_0: A Vision-Language-Action Flow Model for General Robot Control. In *Robotics: Science and Systems (RSS)*, 2025.

Rishi Bommasani et al. On the Opportunities and Risks of Foundation Models. *arXiv preprint arXiv:2108.07258*, 2021.

Siyi Chen, Hugo Hadfield, Alex Zook, Mikaela Angelina Uy, Chan Hee Song, Erwin Coumans, Xuning Yang, Faisal Ladhak, Qing Qu, Stan Birchfield, Jonathan Tremblay, and Valts Blukis. VoLo: A Physical Orchestrator for Open-Vocabulary Long-Horizon Manipulation. *arXiv preprint arXiv:2606.07723*, 2026. URL https://arxiv.org/abs/2606.07723.

Vincent de Bakker, Joey Hejna, Tyler Lum, Onur Celik, Aleksandar Taranovic, Denis Blessing, Gerhard Neumann, Jeannette Bohg, and Dorsa Sadigh. Scaffolding Dexterous Manipulation with Vision-Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 93317–93352, 2025. doi: 10.52202/085713-3121. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/862644b156e51c35dea5a7446d640b14-Abstract-Conference.html.

Kuan Fang, Fangchen Liu, Pieter Abbeel, and Sergey Levine. MOKA: Open-World Robotic Manipulation through Mark-Based Visual Prompting. In *Robotics: Science and Systems (RSS)*, 2024. doi: 10.15607/RSS.2024.XX.062. URL https://www.roboticsproceedings.org/rss20/p062.html.

Letian Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, Wenli Xiao, Fei-Fei Li, Guanya Shi, Jiajun Wu, S. Shankar Sastry, Yuke Zhu, Ken Goldberg, and Linxi Fan. CaP-X: A Framework for Benchmarking and Improving Coding Agents for Robot Manipulation. In *International Conference on Machine Learning (ICML)*, 2026.

Raj Ghugare, Roger Creus Castanyer, Catherine Ji, Kathryn Wantlin, Jin Schofield, Karthik Narasimhan, and Benjamin Eysenbach. BuilderBench: The Building Blocks of Intelligent Agents. *arXiv preprint arXiv:2510.06288*, 2025.

Ankit Goyal, Jie Xu, Yijie Guo, Valts Blukis, Yu-Wei Chao, and Dieter Fox. RVT: Robotic View Transformer for 3D Object Manipulation. In *Conference on Robot Learning (CoRL)*, pp. 694–710, 2023. URL https://proceedings.mlr.press/v229/goyal23a.html.

Asher James Hancock, Xindi Wu, Lihan Zha, Olga Russakovsky, and Anirudha Majumdar. Actions as Language: Fine-Tuning VLMs into VLAs Without Catastrophic Forgetting. In *International Conference on Learning Representations (ICLR)*, 2026.

Hello Robot. Stretch 4: Mobile Manipulator Data Sheet. https://hello-robot.com/wp-content/uploads/2026/05/HelloRobot-DataSheet-Stretch-4-Rev5_AsLaunched.pdf, 2026.

Chi-Pin Huang, Yueh-Hua Wu, Min-Hung Chen, Yu-Chiang Frank Wang, and Fu-En Yang. ThinkAct: Vision-Language-Action Reasoning via Reinforced Visual Latent Planning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2507.16815.

Siyuan Huang, Zhengkai Jiang, Hao Dong, Yu Qiao, Peng Gao, and Hongsheng Li. Instruct2Act: Mapping Multi-modality Instructions to Robotic Actions with Large Language Model. *arXiv preprint arXiv:2305.11176*, 2023a. URL https://arxiv.org/abs/2305.11176.

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 (CoRL)*, 2023b. URL https://arxiv.org/abs/2307.05973.

brian ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander T Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes, Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. In *Conference on Robot Learning (CoRL)*, pp. 287–318, 2023. URL https://proceedings.mlr.press/v205/ichter23a.html.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan P. Foster, Pannag R. Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. OpenVLA: An Open-Source Vision-Language-Action Model. In *Conference on Robot Learning (CoRL)*, 2024.

Yi Li, Yuquan Deng, Jesse Zhang, Joel Jang, Marius Memmel, Caelan Garrett, Fabio Ramos, Dieter Fox, Anqi Li, Abhishek Gupta, and Ankit Goyal. HAMSTER: Hierarchical Action Models for Open-World Robot Manipulation. In *International Conference on Learning Representations (ICLR)*, pp. 24040–24068, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/3bfee3bc6639c36e6e7b058db909f760-Abstract-Conference.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. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2023.

Kevin Lin, Christopher Agia, Toki Migimatsu, Marco Pavone, and Jeannette Bohg. Text2Motion: From Natural Language Instructions to Feasible Plans. *Autonomous Robots*, 2023. URL https://arxiv.org/abs/2303.12153.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. LIBERO: Benchmarking Knowledge Transfer for Lifelong Robot Learning. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023.

Soroush Nasiriany, Fei Xia, Wenhao Yu, Ted Xiao, Jacky Liang, Ishita Dasgupta, Annie Xie, Danny Driess, Ayzaan Wahid, Zhuo Xu, Quan Vuong, Tingnan Zhang, Tsang-Wei Edward Lee, Kuang-Huei Lee, Peng Xu, Sean Kirmani, Yuke Zhu, Andy Zeng, Karol Hausman, Nicolas Heess, Chelsea Finn, Sergey Levine, and Brian Ichter. PIVOT: Iterative Visual Prompting Elicits Actionable Knowledge for VLMs. In *International Conference on Machine Learning (ICML)*, pp. 37321–37341, 2024. URL https://proceedings.mlr.press/v235/nasiriany24a.html.

OpenAI. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*, 2023.

OpenAI. Introducing Codex. https://openai.com/index/introducing-codex/, 2025a.

OpenAI. Introducing Operator. https://openai.com/index/introducing-operator/, 2025b.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. Technical report, OpenAI, 2019.

Lucy Xiaoyang Shi, Brian Ichter, Michael Equi, Liyiming Ke, Karl Pertsch, Quan Vuong, James Tanner, Anna Walling, Haohuan Wang, Niccolo Fusai, Adrian Li-Bell, Danny Driess, Lachy Groom, Sergey Levine, and Chelsea Finn. Hi Robot: Open-Ended Instruction Following with Hierarchical Vision-Language-Action Models. In *International Conference on Machine Learning (ICML)*, 2025. URL https://arxiv.org/abs/2502.19417.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of Bits: An Open-Domain Platform for Web-Based Agents. In *International Conference on Machine Learning (ICML)*, 2017.

Mohit Shridhar, Lucas Manuelli, and Dieter Fox. Perceiver-Actor: A Multi-Task Transformer for Robotic Manipulation. In *Conference on Robot Learning (CoRL)*, pp. 785–799, 2023. URL https://proceedings.mlr.press/v205/shridhar23a.html.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. ProgPrompt: Generating Situated Robot Task Plans using Large Language Models. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2023.

Priya Sundaresan, Hengyuan Hu, Quan Vuong, Jeannette Bohg, and Dorsa Sadigh. What's the Move? Hybrid Imitation Learning via Salient Points. In *International Conference on Learning Representations (ICLR)*, 2025.

Bowen Wen, Shaurya Dewan, and Stan Birchfield. Fast-FoundationStereo: Real-Time Zero-Shot Stereo Matching. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7513–7524, 2026. URL https://nvlabs.github.io/Fast-FoundationStereo/.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2024.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*, 2023.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *International Conference on Learning Representations (ICLR)*, 2024.

Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Kevin Lin, Abhiram Maddukuri, Soroush Nasiriany, and Yifeng Zhu. robosuite: A Modular Simulation Framework and Benchmark for Robot Learning. *arXiv preprint arXiv:2009.12293*, 2020.

Brianna Zitkovich et al. RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control. In *Conference on Robot Learning (CoRL)*, 2023.

A Documentation of MCP Tools

Category	Tools
Observation	`screenshot`, `hover`, `gripper_get_pose`, `gripper_show_rotation_gizmo`, `camera_get_pose`
Action	`gripper_teleport_via_click`, `gripper_drag`, `gripper_translate`, `gripper_advance_or_retreat`, `gripper_rotate`, `gripper_toggle`, `gripper_reset`
Camera	`camera_orbit_via_key`, `camera_pan_via_key`, `camera_zoom`, `camera_reset`
Execution	`execute_waypoint`, `end_episode`

Table 4: **MCP tools.** The agent perceives the interface with `screenshot` and poses the blue target gripper with intuitive tools such as `gripper_teleport_via_click`, `gripper_translate`, and `gripper_rotate`. It can also move the camera via the camera tools to disambiguate the scene. `execute_waypoint` moves the real gripper to the target pose through a controller. Most tools return an updated screenshot reflecting their effect on the UI or the real environment.

The UI described in Section 3.1 already enables a human to control the robot. To make it operable by agents, we wrap its core interactions into a small set of MCP tools, summarized in Table 4 and documented in full below. The tools fall into four categories. *Observation tools* gather information about the environment. `screenshot` returns a screenshot of the entire UI. `gripper_get_pose` and `camera_get_pose` return the robot's proprioceptive state and the current viewport parameters, respectively. We also provide two helper tools. `hover` takes a pair of `(u, v)` pixel coordinates and returns a screenshot with a virtual crosshair overlaid at that location, along with the 3D coordinates of the point it hits in the point cloud, if any. The agent can use it to verify that a planned click will land at the intended location, or to read off the 3D coordinates of points of interest. `gripper_show_rotation_gizmo` overlays a rotation gizmo on the target gripper to help the agent reason about rotation directions. *Action tools* manipulate the target gripper. They have no effect on the real robot until `execute_waypoint` is called. Each action tool returns a screenshot after its change to the target gripper is rendered on the UI, so the agent can check the outcome and keep adjusting until the waypoint is ready for execution. *Camera tools* change the viewport so that the agent can observe the scene from angles beyond the default view. *Execution tools* affect the underlying environment, either moving the robot via `execute_waypoint`, or terminating the episode via `end_episode`.

We design the tools around two principles: *minimalism* and *agent ergonomics*. For minimalism, each tool wraps the corresponding human operation with little or no extra abstraction, so agents can operate the system with the same control and freedom as a human. For agent ergonomics, we account for the fact that agents observe the UI through discrete screenshots rather than continuous animation. Operations such as rotating the gripper by dragging a gizmo axis or orbiting the camera with a continuous drag are therefore hard for an agent to comprehend, so we provide more direct alternatives, such as `gripper_rotate` and `camera_orbit_via_key`, which take exact angles as input.

This appendix includes the detailed signatures and docstrings of all the MCP tools used in this work. These docstrings are accessible to both Claude Code and Codex when they connect to the MCP server. Square brackets in a signature mark optional arguments. All gripper tools except `gripper_toggle` also share a common preamble describing the coordinate frame of the system, reproduced below.

```
Shared coordinate-frame preamble
```

Frame (shared by all spatial tools):

- *robot*: metres; the canonical frame every pose, delta, and camera target reports in and that every tool input expects.

- *orientation*: the gripper's pose is reported as two robot-frame unit vectors, *approach* (the direction the fingers reach) and *opening* (the fingertip-to-fingertip axis the jaws open/close along).

Robot Z position carries a fixed height offset, so a canonical top-down gripper reads $z \approx 1.0$, not 0; reason about position deltas.

A.1 Observation Tools

`screenshot()`

Capture a screenshot of the robot interface. The window shows:

- Left sidebar: two camera feeds. Both are rendered from the real scene, and they refresh only after `execute_waypoint` moves the real gripper. The *third-person view* is a fixed camera looking at the table and gripper, best for analyzing the full scene layout (objects relative to each other and the gripper). The *wrist view* is a camera on the gripper looking down between the fingers, best for the close-up gripper-object relationship (whether the gripper is well positioned for a grasp or place).
- Right main view: a 3D point-cloud reconstruction of the scene, with robot-frame axes (X red, Y green, Z blue) and the blue target gripper mesh you command. Best for 3D and depth understanding that the flat camera images cannot give.

Call this first, or whenever you need a fresh view without taking an action.

`hover(u, v)`

Move the mouse to normalized screenshot coordinates (u, v) and show a visible crosshair overlay in the returned screenshot. The exact (u, v) point is the crosshair's center dot. Use `hover` to inspect a spot before acting on it. It takes no action and gives two readouts:

- Visual: the crosshair (a persistent DOM overlay, not the OS cursor) marks exactly where (u, v) lands, confirming it is on the intended UI element or point-cloud region.
- Contact point (robot-frame $x/y/z$, in metres), computed read-only. On the point-cloud canvas or a camera feed it reports the cloud point under the cursor (the ray need not be pixel-exact; it hits within a small radius). Two uses: probing the robot-frame coordinate of any point of interest, and previewing a teleport, since a point it lands on is exactly the salient point that a `gripper_teleport_via_click` at the same (u, v) would select. Note this is the clicked point itself, not the resulting gripper pose (which lands $\sim$0.07 m back along the approach axis). It reads `none` when (u, v) is not over a cloud point (empty space), over the blue target gripper, or over UI chrome rather than the point-cloud canvas or a camera feed.

`gripper_get_pose()`

Return the current blue target gripper pose and visually show its approach axis in the screenshot. Besides giving a readout of the exact current gripper pose, the drawn approach axis makes this the best orientation check before using `gripper_advance_or_retreat`: it shows which way `f`/`b` will step. Reports the center of the fingertips of the blue target gripper in robot-frame metres, its orientation as approach and opening unit vectors, and the open/closed state. The visual axis guide is shown without moving the gripper.

`gripper_show_rotation_gizmo()`

Show the gripper's rotation gizmo (the three rotation rings) together with its approach axis in the screenshot. Always call this first to choose the `gripper_rotate` axis: it gives an intuitive visual understanding of the rotation axes (the x/y/z gizmo rings). The arrow on each rotation

ring points in the + direction of that rotation. Reports the center of the fingertips of the blue target gripper in robot-frame metres, its orientation as approach and opening unit vectors, and the open/closed state.

`camera_get_pose()`

Return the current camera pose without taking an action. The camera orbits a target point, reported in the robot frame (metres) like every other pose: `target` is the world point the camera looks at, and `distance` is the camera-to-target separation in metres. `azimuth` and `elevation` give the viewing direction onto that target: azimuth is the horizontal orbit angle in degrees; elevation is 0 at horizontal (level with the target) and 90 at top-down, but orbiting keeps it within $[0, 85]$ so it never reaches the degenerate top-down pole (where azimuth would collapse to 0). The target is reported because pan and off-center zoom shift the scene center. Use these values to reason about the current framing before adjusting it with other camera tools.

A.2 Action Tools

`gripper_teleport_via_click(u, v)`

Fast-move the blue target gripper near a clicked salient point; returns a screenshot. Best for quickly setting a subtask's starting position; refine afterward with `gripper_drag`, `gripper_translate`, `gripper_rotate`, or `gripper_toggle`. Where to click:

- The left-panel 2D camera views are best when the target is clearly visible in either camera feed.
- The right-panel 3D point-cloud view is best when locating the target takes some 3D reasoning about depth.
- Click the object's center, not its edge, so the gripper lands centered over it for a reliable grasp or place.

The gripper lands $\sim$0.07 m back from the clicked point along its current approach axis (opposite the reach direction), not on the point itself, so a short forward move or descent then reaches it. For example, if the gripper currently points straight down, this puts it $\sim$0.07 m directly above the click. Translation only: the gripper's orientation is preserved. The readout reports the clicked point as the "selected salient point" and the resulting gripper pose separately; check both, plus the screenshot. Clicking the blue target gripper itself does nothing here; that is reserved for `gripper_drag`.

`gripper_drag(u1, v1, u2, v2, [constraint], [steps])`

Drag the blue target gripper from (u_1, v_1) to (u_2, v_2); returns a screenshot. The blue target gripper is the target pose you command; this tool moves it. The gray point-cloud gripper is actual sensor data and is not a draggable mesh. The start point (u_1, v_1) must ray-hit the solid blue target gripper mesh (a finger, wrist, or body), not the empty gap between the fingers or the gray gripper; a miss is refused with a no-op. A free drag moves the gripper with the cursor in the view plane, $\sim$0.0006 m per CSS pixel of mouse movement, before constraints. The optional `constraint` picks the movement mode; x/y/z lock motion to one robot-frame axis (the fixed robot X/Y/Z, not the gripper-attached gizmo axes that `gripper_rotate` spins about) and are exact at any camera angle, so trust the pose readout, not the picture:

- `free` (default): unconstrained, follows the cursor in the camera view plane (camera-dependent).
- `x`: lock to robot X only.
- `y`: lock to robot Y only.
- `z`: lock to robot Z (vertical).

Tip: `hover` first to confirm (u_1, v_1) lands on the gripper, then use small single-axis drags. The screenshot shows start and end markers, the end crosshair, the arrow path, and the updated pose. The optional `steps` sets the number of mouse-move interpolation steps (default 25).

`gripper_translate([d_x], [d_y], [d_z])`

Translate the blue target gripper by a metric offset along the robot-frame axes, in metres; returns a screenshot. You specify the offset directly, with no start point, mesh hit, or mouse drag needed, so it is the exact, camera-independent way to shift the gripper along the world axes. This is the keyed equivalent of `gripper_drag`'s x/y/z axis locks without the cursor imprecision; the orientation is unchanged. Axes are the fixed robot X/Y/Z (the colored arrows), not the gripper-attached gizmo axes that `gripper_rotate` spins about, so the offsets are exact at any camera angle: `d_x` moves along robot X, `d_y` along robot Y, and `d_z` along robot Z (vertical: $+z$ lifts, $-z$ lowers). Set one offset for a pure single-axis move, or several to move diagonally; omitted offsets default to 0 and at least one must be nonzero. Each offset must be within ± 0.1 m to keep moves gradual; larger requests are rejected, so call repeatedly to translate further. The returned text reports the actual pose delta after the move.

`gripper_advance_or_retreat(direction, [steps])`

Move the blue target gripper along its approach axis; returns a screenshot. Direction `f` steps along the approach axis (toward where the fingers point); `b` steps opposite it (retreat, back out). In the robot frame the direction follows the current orientation: at a top-down pose, `f` is $-Z$ (down) and `b` is $+Z$ (lift), while on a side grasp the same steps are horizontal. If the current orientation makes the direction ambiguous, call `gripper_get_pose` first to see the approach axis drawn in the screenshot. One step is about 0.006 m along the approach axis; use 1–3 steps for small corrections and larger counts for coarse approach or retreat (default 1). The returned text reports the actual pose delta.

`gripper_rotate(axis, angle)`

Rotate the blue target gripper about one gizmo axis (`x`, `y`, or `z`) by a signed angle in degrees; returns a screenshot. Important: call `gripper_show_rotation_gizmo` before every rotation and pick the axis and direction from the gizmo you see; do not guess. The x/y/z here are the gripper's own gizmo axes, not the world/robot X/Y/Z (the colored arrows); they line up only at the top-down pose and move with the gripper after any tilt, so the same letter points a different way once the gripper is tilted. The gripper has three mutually perpendicular gizmo axes: the approach axis (where the fingers reach), the opening axis (the finger line), and a third axis perpendicular to both. Each rotation spins about one, leaving it fixed and re-aiming the other two (the arrow on each ring shows the $+$ direction):

- `x`: about the third axis; both the approach and opening axes re-aim.
- `y`: about the opening axis; opening stays fixed, approach re-aims.
- `z`: about the approach axis; approach stays fixed, opening re-aims (twist in place).

The angle must be within $[-90, 90]$ degrees; prefer small steps (10–45 degrees) so you can observe the intermediate result and adjust before overshooting. The returned text reports the resulting approach and opening axes; the rotation is in place (position unchanged). If you are unsure about the axis or direction, rotate by a small amount and observe the outcome.

`gripper_toggle([state])`

Open or close the blue target gripper; returns a screenshot. Wraps the UI `g` key, which swaps the gripper mesh between open and closed. Pass `state` to drive a known result without checking the current state first: `toggle` (default) flips the current state, `open` ensures open (no-op if

already open), and `closed` ensures closed (no-op if already closed). The returned message and screenshot report the resulting open/closed state, read back after the swap.

`gripper_reset()`

Reset the blue target gripper back onto the real gripper's current pose (position, orientation, and open/closed state); returns a screenshot. Use this to undo experimental edits: freely try `gripper_drag`, `gripper_advance_or_retreat`, `gripper_rotate`, or `gripper_toggle` to plan a move, see how it looks, then reset to the real pose before committing the move you actually want. This only moves the blue target gripper; it does not move the real robot and records no waypoint.

A.3 CAMERA TOOLS

`camera_orbit_via_key(d_azimuth, d_elevation)`

Orbit the camera around the scene target by an angular delta, in degrees; returns a screenshot. You specify the two orbit angles directly, with no start point or open-space click needed, so it is the simplest way to reframe. Both deltas are relative to the current pose (read `camera_get_pose` for the absolute angles); the distance and look-at target are unchanged, as this only rotates the viewpoint. Positive `d_azimuth` swings the camera counterclockwise seen from above, negative swings it clockwise; at the default view, azimuth is 90 with the camera at $Y = 0$ looking toward $-X$, and positive deltas swing it toward $+Y$, reaching a view toward $-Y$ at azimuth 180. Positive `d_elevation` raises the camera toward a top-down view (elevation toward 90); negative lowers it toward a front or side horizontal view (elevation toward 0). Each delta must be within ± 45 degrees to keep camera moves gradual; larger requests are rejected, so call repeatedly to orbit further. Elevation is clamped to $[0, 85]$ (the horizon is allowed; the exact top-down pose is degenerate and kept just out of reach); azimuth wraps at 360. Returns the updated camera pose.

`camera_pan_via_key(d_right, d_up)`

Pan the camera (shift its look-at point) by a metric offset, in robot-frame metres; returns a screenshot. You specify the shift directly, with no start point or drag needed, so it is the simplest way to re-center the view without rotating. The viewing angle (azimuth/elevation) and distance are unchanged: the camera and its target slide together by the same amount, so only the framing moves. Positive `d_right` pans right, sliding the look-at point toward screen-right so you see more of what was off the right edge (scene content appears to slide left); negative pans left. Positive `d_up` pans up along world-Z (robot $+z$) so you see more above (scene content appears to slide down); negative pans down. Each offset must be within ± 0.2 m to keep moves gradual; larger requests are rejected, so call repeatedly to pan further. Returns the updated camera pose, whose target shows the new framing center.

`camera_zoom(u, v, [steps])`

Zoom the camera toward normalized screenshot coordinates; returns a screenshot. The browser zooms toward the point under the mouse; (u, v) should be inside the point-cloud view. Positive `steps` zoom in and negative zoom out (default 1); internally this sends scroll-wheel events to the browser. Each step changes the camera-to-target distance by a constant $\sim$0.03 m, regardless of the current zoom level or direction, so the step count is roughly the desired distance change divided by 0.03 m. An off-center zoom also shifts composition toward that point; use this deliberately to reframe while zooming. Returns the updated camera pose.

`camera_reset()`

Restore the initial camera view; returns a screenshot and the updated camera pose. Strategy: the default view is framed so most operations are viable from it, so prefer working there and return to it with this tool after exploring. Reach for `camera_orbit_via_key` or `camera_pan_via_key` only when the default view is insufficient, e.g., to resolve an occlusion or to judge the gripper-to-object relationship (depth, alignment, clearance) from another angle, then reset here to re-establish a known frame.

A.4 Execution Tools

`execute_waypoint()`

Execute the current gripper target and record one waypoint; returns a screenshot. This asks the robot controller to move the real gripper to the current target pose: position, orientation, and final open/closed gripper state. Use this after meaningful target-setting operations such as `gripper_teleport_via_click`, `gripper_drag`, `gripper_advance_or_retreat`, `gripper_rotate`, or `gripper_toggle`. Execute a `gripper_toggle` on its own; do not combine it with a pose change in the same waypoint, since moving and opening/closing the gripper at once is unreliable. Execution uses the final target as the next base state for later operations. The controller interpolates between the current pose and the target pose; it does not replay the exact UI trajectory used to place the target. For example, if you toggled the gripper twice before executing, it only tries to match the final open/closed state, without intermediate toggles. If this waypoint achieves the task goal, the result reports `TASK SUCCEEDED` and that the connection is about to close. When you see that, stop: the episode is done and no further tools should be called.

`end_episode()`

Save and end the current episode and reset or advance the task; returns a screenshot. Call this only after the final waypoint for the episode has been executed.

B TOOL CALL EFFICIENCY AND ESTIMATED COSTS

	Claude Code			Codex		
Task	Medium Sonnet 5	Large Opus 4.8	Extra Fable 5	Medium Terra 5.6	Large Sol 5.6	Extra Astra 6
Turn on stove	84	36	39	46	48	**16**
Open drawer	117	47	38	91	33	**19**
Put bowl on plate	174	59	53	72	52	**34**
Square	81	92	69	–	87	**42**
Rainbow	271	159	143	175	208	**136**
T-block	–	81	89	162	109	**59**
Overall	146	79	72	109	90	**51**

Table 5: **The average number of tool calls per successful episode.** Mean number of MCP tool calls over successful episodes for the default-prompt evaluations in Table 1. '–' indicates that no episode succeeded. Bold marks the fewest tool calls in each row. The Overall row averages task means over tasks with at least one successful episode.

	Claude Code			Codex		
Task	Medium Sonnet 5	Large Opus 4.8	Extra Fable 5	Medium Terra 5.6	Large Sol 5.6	Extra Astra 6
Turn on stove	$2.8	$4.1	$8.9	$1.1	$2.5	**$1.0**
Open drawer	$4.8	$7.4	$7.5	$2.8	**$1.0**	$1.3
Put bowl on plate	$5.8	$7.5	$11.8	$1.9	**$1.7**	$2.5
Square	**$2.8**	$12.5	$14.1	–	$3.5	**$2.8**
Rainbow	$10.2	$26.0	$36.1	**$1.6**	$4.5	$7.6
T-block	–	$9.7	$19.2	$4.9	$3.8	**$3.7**
Overall	$5.3	$11.2	$16.3	**$2.4**	$2.8	$3.2

Table 6: **The average API cost per successful episode.** Estimated mean API cost in USD over successful episodes for the default-prompt evaluations in Table 1. Costs are estimated from recorded token usage and the pricing assumptions stored with the analysis. '–' indicates that no episode succeeded. Bold marks the lowest cost in each row, including ties at the reported precision. The Overall row averages task means over tasks with at least one successful episode.

We report the mean number of MCP tool calls in Table 5 and the average API cost in Table 6, both over successful episodes. We omit failed episodes because the agents often keep trying until the one-hour limit, which would significantly skew the numbers and make them highly correlated with the success rates.

C PERFORMANCE NUMBERS OF THE BASELINE

	Claude Code		Codex		
Tier	Large	Extra	Medium	Large	Extra
Task	Opus 4.8	Fable 5	Terra 5.6	Sol 5.6	Astra 6
Turn on stove	0%	50%	0%	10%	100%
Open drawer	10%	20%	10%	30%	80%
Put bowl on plate	0%	20%	10%	40%	100%
Square	10%	20%	10%	20%	80%
Rainbow	20%	20%*	10%	50%	100%
T-block	0%†	10%	0%	0%	100%
Overall	6.7%†	23.3%*	6.7%	25.0%	93.3%

Table 7: **Minimal-prompt scaling results for the naive baseline.** Success rates over ten seeds per task using Claude Code and Codex. The Overall row averages all six tasks equally. An asterisk marks a provisional Rainbow grade and aggregates that depend on it; a dagger marks the Opus 4.8 T-block run, which was not executed and is counted as 0%.

We report per-task success rates for the naive baseline in Table 7, complementing the aggregate results in Fig. 5. The baseline uses the same observations, agent, and controller as VIA, but sends commands directly to the controller without staging or refining them in the virtual workspace. It has only five MCP tools: `screenshot`, `get_proprio`, `set_pose`, `set_gripper`, and `end_episode`. `set_pose` and `set_gripper` immediately invoke the controller to move the end-effector to a specified 6-DoF pose and set the gripper's open/closed state, respectively. `screenshot` returns the same observation as the corresponding tool in VIA. `get_proprio` returns the robot's proprioceptive state, and `end_episode` terminates the episode.

D REAL-WORLD ROBOT PLATFORM



Figure 7: **Real-world robot platform.** The Stretch 4 mobile manipulator used for the five real-world tasks in Section 4.3.

Stretch 4 is a recently introduced mobile manipulation platform with a 3-DoF omnidirectional base, a 1-DoF vertical lift along the torso, a 1-DoF telescoping arm, and a 3-DoF wrist (Hello Robot, 2026) (Fig. 7). We use its head camera, which is fixed relative to the base, for visual observations and its head-mounted LiDAR sensors for scene-level depth sensing. The robot also has a wrist-mounted stereo camera, from whose images we compute depth using Fast-FoundationStereo (Wen et al., 2026).

E PROMPTS

E.1 SYSTEM PROMPT

For the simulation experiments, the roughly 650-word system prompt covers the basics of the interface (target gripper, waypoints, camera feeds, point cloud workspace) and general guidance on using it effectively (task decomposition, closed-loop verification, common failure modes such as grasping too high). We reproduce it below, reformatted from Markdown into LaTeX with the content unchanged, apart from redacting internal project names and file paths.

Robot Control Guide

You operate a simulated robot arm through the MCP server to complete manipulation tasks.

General Information. You control the robot via a 3D browser GUI, similar to 3D design software or a 3D game. The scene contains a robot arm with a gripper and a tabletop environment where you perform manipulations as instructed.
You control the robot by setting and executing a sequence of "waypoints":

- There is a blue virtual gripper in the scene, called the "target gripper". You manipulate this target gripper via tools to set a "waypoint".
- Repeat the following to complete a task, one waypoint at a time:
  - Set: move the target gripper to a desired pose (waypoint) via the tools defined in the MCP server.
  - Execute: A controller will bring the real gripper to the target gripper via a linear interpolation while obeying physical constraints.
  - Check: After the waypoint is reached, the tool will return an image showing the updated UI (outcome). Decide the next pose based on the outcome. If the outcome isn't what you intended, diagnose why it failed and retry that step or any earlier one that caused it.

For example, to pick up a cube: move the gripper above it, descend until the cube sits between the fingers, close, then lift — each step is one waypoint that you set and then execute before moving on to the next.

Tips: Decomposition and Planning.

- Decompose the task into **small subtasks** by breaking the goal into a short sequence of simple stages.
- Prefer closed-loop control over open-loop. Adjust the plan as you observe the outcome of previous waypoints.
- Give `gripper_toggle` its own `execute_waypoint` cycle. Do NOT combine it with another pose change (translate/rotate/advance...) in one waypoint. Moving and toggling the gripper at the same time could be unpredictable.

Tips: Precision.

- Be precise with `gripper_teleport_via_click` — it's the best way to set a good initial position.
  - The clicked point lands on the gripper's approach axis, so a precise click will make future approaching easy.
  - Use `hover` to double-check whether the predicted (u, v) would land where you intend to click.
- Placing an object onto another object is a precision-sensitive task. First reason about where the held object should land, and then reason about where to set the target gripper, accounting for the relation between the gripper and the held object.
- Always check the visual from the point cloud or the camera feeds. Do NOT solely rely on the coordinates from `hover`. When necessary, make fine adjustments with translate, rotate, advance_or_retreat, and similar operations.

Tips: Viewing and Camera Movement.

- Two camera feeds on the left.
  - They provide raw camera images from 3rd-person view and wrist view.

  - The 3rd-person view is good for understanding the scene and disambiguating objects.
  - The wrist view is good for close-up analysis and fine adjustments.
  - `gripper_teleport_via_click` also accepts clicking on a camera feed. Sometimes this could be more convenient than clicking on the point cloud.
- The 3D point cloud on the right.
  - This is the main workspace. It constructs the 3D scene from multiple depth cameras.
  - The initial view (resettable via `camera_reset`) is framed to be good for most operations. It is a good default so prefer working in it.
  - A pure top-down view (elevation ≥ 80) may not be as useful as you think, because the gripper and robot arm may block most of the workspace.
  - Sideview (orbit by changing azimuth ± 30–45) or frontview (elevation ~ 10) could be very helpful if object-gripper or object-object relation is hard to interpret in the initial default view.
  - Move the camera gently, and reset to the initial view via `camera_reset` to recover from a failed camera operation.

Tips: Common Failures.

- Grasping too high is a common failure. Descend deep so that the fingers and the object to hold have enough contact surface.

For the real-robot experiments on the Stretch 4 mobile manipulator (Section 4.3), the agent receives a different system prompt that describes the robot's kinematics and coordinate frames, the 2D interface with head and wrist camera feeds and a virtual target gripper overlay, and the three waypoint types (navigation, pose, and gripper). We reproduce it below, reformatted from Markdown into LaTeX with the content unchanged, apart from redacting internal project names and file paths.

Robot Control Guide (Stretch 4)

You operate a Stretch4 robot through the MCP server to complete navigation and manipulation tasks.

Robot Information. The robot has

- An omnidirectional mobile base: it can translate in any horizontal direction and rotate in place.
- A vertical torso fixed to the base; the arm rides up and down along it (robot z-axis).
- A horizontal arm that telescopes (extends or retracts) along its own axis (robot x-axis).
- A parallel-jaw gripper with jaw aperture 0–8 cm on a 3-DOF wrist at the end of the arm.

Axes. We have two sets of axes that are handy for different tools:
Robot-centric axes:

- x: forward — where the arm points and extends.
- y: left — 90 degrees counter-clockwise from x.
- z: up — along the vertical torso.

The robot axes are attached to the base and rotate with the base. They are handy for reasoning about navigation, as well as coarse alignment via base rotation and translation.
Wrist-centric axes:

- x: forward — the approach axis, where the fingers point.
- y: left — the opening axis, the line the jaws open and close along.
- z: up — perpendicular to both x and y, through the jaw center.

The wrist axes ride and rotate with the gripper. For example, an `ee_wrist_centric_rotate` about the x-axis will change both the y- and z-axes. They match the robot axes only when the gripper is level and pointing forward. They are handy for reasoning about fine adjustments via `ee_wrist_centric_translate` / `ee_wrist_centric_rotate`.

Control Interface. You control the robot to perform tasks by observing through its cameras and commanding it through a sequence of waypoints using the MCP tools, similar to 3D design software or a 3D game.

Observation:

- The observation is a collection of images and proprio status.
- The images are from a head camera (headview) and a wrist camera (wristview).
- The head camera:
  - Fixed relative to the robot base.
  - Moves and rotates with the base.
  - Good for scene understanding, navigation, long-distance reaching.
  - The head view is about 65 degrees wide. When scanning for something, rotate at most 45 to 60 degrees per step so consecutive views overlap.
- The wrist camera:
  - Mounted next to the gripper.
  - Moves with the base as well as the arm.
  - Good for fine adjustment and grasp analysis.
- Both views also contain a virtual **target gripper**:
  - In headview, it is a red gripper overlay.
  - In wristview, it is a red `|----.----|` glyph.
  - Every tool that moves the target gripper will return views with an updated rendering of the virtual gripper, reflecting where the real gripper will go without moving the real robot.
  - It is the main medium for reasoning and planning your waypoints before executing them.

Waypoint:

There are 3 types of waypoints:

- nav:
  - Issued by one or more calls to `nav_translate` and `nav_rotate`.
  - Moves the omni base only and does not change the pose of the end-effector relative to the base.
  - Good for navigation, long-distance movement, or searching for objects of interest.
  - Be careful to avoid collisions when navigating.
- pose:
  - Issued by one or more calls to `ee_{...}` tools.
  - Moves the end-effector to the desired pose via a whole-body controller that moves both the base and end-effector.
  - Good for object-centric movement (`ee_teleport_via_click`), or close-up manipulation (`ee_robot_centric_translate`, `ee_wrist_centric_translate`, `ee_wrist_centric_rotate`).
- gripper:
  - Issued by `gripper_set`, setting the gripper to open or closed.

More about waypoints:

- For simplicity, a waypoint can only be in one of the three modes.
- Within the same mode, multiple tool calls can be made to achieve a desired target.

Workflow.

- `observe`: the initial call to understand the scene and come up with a plan of waypoints.
- Repeat until the task is complete, one waypoint at a time:
  - Adjust the **target** in a mini-loop: call a tool, inspect the returned views, and call again until the virtual gripper sits where you want it. No real motion happens in this loop.
  - `execute_waypoint`: a controller will bring the real robot to the target pose via linear interpolation while obeying physical constraints.
  - Check-and-adjust: after the waypoint is reached, analyze its real outcome and adjust the plan for the subsequent waypoints if needed.
- `end_episode` when you believe the task is done, or has become unachievable.

For example, to pick up a cube:

- Locate the cube in the headview.

- Move close to it.
- Descend and re-orient the gripper until the cube sits between the fingers.
- Close the gripper.
- Lift.

Each step is one waypoint that you set and then execute before moving on to the next.

Tips.

- `hover` returns the offset of the hovered point (when available) using depth, but the depth is noisy and the depth from the wrist and head cameras may not agree, especially when the object is far away or off-center. Therefore, always trust the images and do not over-rely on the offset information from `hover`.
- A common failure is grasping short: closing while the object is still in front of the finger pads, or barely at their tips. Make sure that the fingers and the object have **enough** contact area. Advance until the intended contact sits deep between the pads before closing. With open jaws, slightly overshooting is usually safer than undershooting.

E.2 Task Prompts

Each episode's prompt has two parts: a one-sentence task instruction embedded in a fixed user-message template, and a task-specific tips file appended to the agent's system prompt. For the three LIBERO-Goal tasks, Square, and T-block, we provide both *default* and *detailed* prompts; the detailed versions contain example lists of waypoints. Rainbow has only a default prompt. We reproduce all of them below, reformatted from Markdown into LaTeX with the content unchanged.

Turn on stove.
Task instruction: "Turn on the stove."

Turn on stove: default prompt

The goal is to turn on the stove, which requires rotating the black control knob on the back of the stove by 90 degrees counter-clockwise viewed from the top.
Think carefully about what the task actually requires and about the tools available to you. Plan an approach that reaches the goal with precision and efficiency.

Turn on stove: detailed prompt

To turn on the stove, we need to rotate the black control knob on the back of the stove by 90 degrees counter-clockwise.
Example waypoints:

1. Locate the black control knob on the back of the stove and use `gripper_teleport_via_click` to teleport the gripper on top of the knob.
2. Lower the gripper towards the knob, and adjust the gripper if needed so that the stem of the knob is roughly in the middle of the fingers.
3. Close the gripper to hold the stem of the knob.
4. Rotate the gripper 90 degrees counter-clockwise to turn the stove on.

Open drawer.
Task instruction: "Open the middle (2nd from the top) drawer of the cabinet."

Open drawer: default prompt

The goal is to open the middle (2nd from the top) drawer of the cabinet by pulling it out by its handle.
Think carefully about what the task actually requires and about the tools available to you. Plan an approach that reaches the goal with precision and efficiency.

Open drawer: detailed prompt

The trick to opening a drawer is to **first** rotate the gripper so that the approach axis will point towards the handle when we use `gripper_teleport_via_click`. Then clicking the handle lands the gripper $\sim$0.07 m directly in front of it, already aligned to drive straight in.
Example waypoints:

1. Rotate the gripper so its approach axis points roughly horizontally at the handle.
2. Use `gripper_teleport_via_click` on the **handle** of the **correct** drawer.
3. Move forward along the approach axis until the handle is between the open fingers. Check in the wrist view that the handle is centered. Use `gripper_translate` or `gripper_rotate` to center the handle between the jaws if it is off.
4. Close the gripper.
5. Pull straight back along the approach axis, keeping the gripper level, until the drawer is open (may take more than one pull).

Put bowl on plate.
Task instruction: "Put the bowl on the plate."

Put bowl on plate: default prompt

The goal is to pick up the bowl and place it on the plate. Success requires the bowl to end up resting at the **exact center** of the plate.
Think carefully about what the task actually requires and about the tools available to you. Plan an approach that reaches the goal with precision and efficiency.

Put bowl on plate: detailed prompt

How to pick up a bowl. To pick up a bowl, we need to pinch it by its side wall with a straight top-down descent. Here we give example waypoints assuming we pick it from the right ($+Y$ direction).
Example waypoints:

1. Locate the bowl, and click on the rightmost point on the rim (pinch point). Trick: The click point **must** land on the bowl not on the table; after `gripper_teleport_via_click`, read the salient point's Y from the output; nudge the click along the rim and re-check until Y stops increasing while the click point still lands on the bowl not on the table.
2. Descend the gripper low for a firm grasp.
3. Close the gripper.
4. Lift the gripper. After a lift, judge grasp success from the point cloud (frontview could be helpful), not the wrist view.

How to place onto a plate. This task requires the held object to be placed **exactly at the center** to claim a success. When holding a bowl, the held object's center is offset from the gripper, so we need to account for that when deciding the target location for the gripper.
Example waypoints:

1. `gripper_teleport_via_click` on the desired location of the plate so that the held object will roughly appear above the center of the plate. E.g. holding the bowl by its right ($+Y$) rim, click on the right ($+Y$) half of the plate. Adjust the height after clicking to avoid collision.
2. Observe the outcome of the first waypoint and adjust the gripper if needed. Do NOT solely trust the readout from `hover`. Check the visual from the camera and 3D point cloud carefully as well.
3. Descend until the object rests on the plate (contact stops the descent), with a final small XY nudge if needed.
4. Open the gripper to release the held object.

Square.
Task instruction: "Pick up the square nut and place it onto the square peg."

Square: default prompt

The goal is to pick up the square nut and place it onto the square peg (not the round peg). Success requires the square peg inside the nut's hole and the nut slid all the way down so it rests near the table.
Think carefully about what the task actually requires and about the tools available to you. Plan an approach that reaches the goal with precision and efficiency.

Square: detailed prompt

How to pick up the square nut:
Pinch the nut's **projecting tab** from above, leaving the square hole clear for the peg.
Example waypoints:

1. Use `gripper_teleport_via_click` on the exposed tab, where both fingers can descend beside it without hitting the ring.
2. With the gripper pointing down, align the jaws **across the tab's narrow width**. Center the tab between them and descend deep enough to straddle its thickness.
3. Close the gripper.
4. Lift. Verify in the point cloud or 3rd-person view that the whole nut rises with the gripper.

How to place onto the square peg:
Center the **hole** over the square peg, accounting for its offset from the gripper. Rotating the held nut changes this offset.
Example waypoints:

1. Lift until the nut's bottom clears the peg before moving sideways. Keep it level and rotate until the hole's sides are parallel to the peg's sides.
2. Click the peg's top face, then adjust for the gripper-to-hole offset and height before executing. Check the actual hole-over-peg alignment in the cameras and point cloud.
3. Descend in small steps, watching the peg enter the hole. If the nut catches or tilts, lift clear and correct the alignment.
4. Keep holding the tab as the nut slides down. Once it rests near the table with the peg passing through its hole, open the gripper.

Rainbow.
Task instruction: "Rearrange the blocks so that it forms a rainbow. Be creative."

Rainbow: default prompt

Note that this is a creativity task with no built-in reward/termination detector. Call `end_episode` when you are done.
You may accidentally knock over some blocks while arranging others, therefore you should double check the final arrangement before calling `end_episode`.

T-block.
Task instruction: "Build a structure exactly two blocks tall: the blue block on the bottom, and both green blocks resting directly on top of it, side by side at the same height. Neither green block may sit on the other or on the table."

T-block: default prompt

The scene has three cubes of the **same size**: one blue and two green.
Goal: build a structure that is exactly **two cubes tall**:

- Bottom layer: the blue cube, resting on the table.
- Top layer: **both** green cubes, sitting side by side on top of the blue cube.

For the result to count as correct, **all** of the following must hold:

- Both green cubes rest **directly** on the blue cube — each green cube touches the blue cube's top face.
- The two green cubes are at the **same** height (same level), side by side.
- The final structure is **two cubes tall**, never three.
- The final structure is stable: each green cube has enough contact/support area on the blue cube that it will not tip or fall.

Each cube is the same size, so two greens side by side are twice as wide as the blue:

```
[green][green]   <- top layer: both greens, same level, both on the blue
   [blue ]       <- bottom layer: blue on the table (same size as one green)
==============      table
```

Incorrect (do NOT do this):

- A three-tall tower: one green on the blue, and the second green stacked on top of the first green.
- Either green cube resting on the table, or on the other green cube, instead of directly on the blue cube.
- The two green cubes ending up at different heights.

Tips: Note that because each green cube is exactly as wide as the blue cube's top face, two greens placed side by side in the blue's default orientation cannot both get enough support area — at least one will overhang too far and tip off. Reason about how to reorient the blue cube so that both greens gain enough contact area to rest stably, side by side, at the same level.

T-block: detailed prompt

The scene has three cubes of the **same size**: one blue and two green.
Goal: build a structure that is exactly **two cubes tall**:

- Bottom layer: the blue cube, resting on the table.
- Top layer: **both** green cubes, sitting side by side on top of the blue cube.

For the result to count as correct, **all** of the following must hold:

- Both green cubes rest **directly** on the blue cube — each green cube touches the blue cube's top face.
- The two green cubes are at the **same** height (same level), side by side.
- The final structure is **two cubes tall**, never three.
- The final structure is stable: each green cube has enough contact/support area on the blue cube that it will not tip or fall.

Each cube is the same size, so two greens side by side are twice as wide as the blue:

```
[green][green]   <- top layer: both greens, same level, both on the blue
   [blue ]       <- bottom layer: blue on the table (same size as one green)
==============      table
```

Incorrect (do NOT do this):

- A three-cube-tall tower: one green on the blue, and the second green stacked on top of the first green.
- Either green cube resting on the table, or on the other green cube, instead of directly on the blue cube.
- The two green cubes ending up at different heights.

How to prepare the blue base:

- Rotate blue **45 degrees around the z-axis** so both green cubes can keep their centers over blue's top face.

Example waypoints:

1. Click blue's top-face center with `gripper_teleport_via_click`, descend for a deep grasp, then close.
2. Lift, rotate about 45 degrees, and lower it flat onto the table.
3. Open and lift clear.

How to place the green cubes:

- Viewed from above, the greens should sit side by side on blue: one on the left and one on the right, with a minimal gap.
- Use `hover` to sample points along the **front-to-back diagonal of the blue cube's top face**, including its two opposite corners and midpoint.
- This line divides the support area between the greens: place one green on each side.

Example waypoints:

1. Grasp the left green deeply and centrally; lift it level. Position it on the left half of blue, with its **right side parallel and as close as possible to the diagonal without crossing it**. Center that side along the diagonal.
2. Lower onto blue, release, and lift straight up. Verify the green's top stays one cube height above blue's; if it tips, retry closer to the diagonal.
3. Grasp the right green from **front and back**, with the jaws opening parallel to the diagonal, so neither finger hits the left green during placement.
4. Keep the right green's bottom above the left green's top while moving over blue's right half. Align its **left side parallel and as close as possible to the diagonal without crossing it**, centered along the line. Lower vertically, leaving a minimal gap between the greens.
5. Release and lift clear. Verify both greens remain stable at the same height on blue.