SpaceTools: Tool-Augmented Spatial Reasoning via Double Interactive RL

Supplementary Material

We provide additional details and extended results in the supplementary materials,

- Appendix A: Limitations and future directions.
- Appendix B: Further details on the Toolshed system.
- Appendix C: Expanded method descriptions.
- Appendix D: Additional implementation details.
- Appendix E: Extended experimental results.

A. Limitations and Future Directions

Our work shows that tool-augmented spatial reasoning, enabled through DIRL and the Toolshed infrastructure, provides an effective and scalable foundation for training VLMs with strong spatial reasoning, robust tool coordination, and broad generalization across diverse tasks and embodiments. At the same time, this framework opens several promising directions that fall beyond our present scope but merit deeper exploration. We discuss these limitations and future opportunities below.

Application scope. A natural next step is to broaden the range of tasks and environments in which tool-augmented spatial reasoning is applied. Our current scope focuses on short- or medium-horizon tasks, such as spatial question answering or grasp-and-place manipulations. Extending to more complex, longer-horizon, or multi-stage tasks may further reveal the potential of tool-augmented reasoning, allowing the model to concentrate on reasoning and decision-making rather than learning numerous precise perceptual subtasks. Moreover, integrating richer environments, including large-scale robotic simulation, interactive game environments, or physics-rich virtual worlds, could support more diverse experiences and ultimately more general embodied spatial intelligence.

Methodology. From a methodological perspective, several directions could strengthen the flexibility and robustness of tool-augmented spatial reasoning. Although Toolshed supports image-level tool outputs, this work primarily explores tools that return structured text or variables (*e.g.*, point cloud). Extending the model to reason over visual outputs from tools may unlock more expressive or fine-grained reasoning behaviors. Another important direction is systematically improving how VLMs perceive, verify, and recover from tool errors or inaccuracies. Moreover, under a modular perspective, future work could investigate enhancing particular system capabilities by upgrading a single tool without modifying other components, while ensuring that overall tool coordination remains robust. Additionally, while our RL exploration focuses on prompt design, loss functions, and task-specific reward formulations, alternative RL approaches, such as stepwise reward formulations, may improve learning effectiveness in large multi-tool action spaces. Finally, continual learning of new tools is also an important future direction.

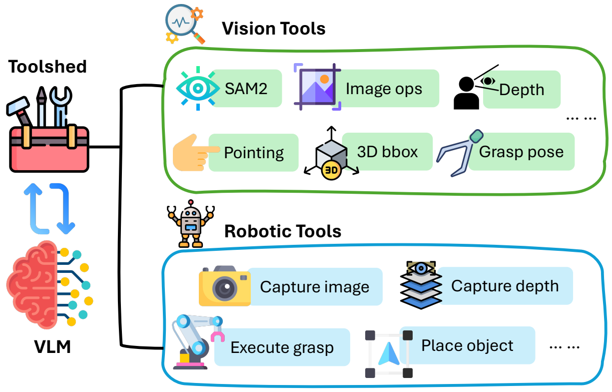


Figure 1. The **Toolshed** infrastructure linking a VLM with modular vision and robotic tools under a unified toolbox for perception and control.

Infrastructure. On the system side, Toolshed provides a scalable backbone for interactive tool use and learning, but there remains room to further improve its efficiency and resource utilization. Serving many heterogeneous tools in parallel can introduce latency and memory bottlenecks, particularly for high-resolution vision tools or robot-in-the-loop executions. In this work, we mitigate the latter by using mock robot tools during training. Future advances in scheduling, caching, batching, and asynchronous execution could potentially enhance performance and even support real robot execution effectively during interactive learning. Additionally, developing lighter-weight tools, model-side approximators, or memory-optimized deployment strategies may reduce overhead and enable larger-scale training or more complex task environments. These improvements would allow the framework to scale more gracefully as tool diversity and task complexity increase.

B. Details of Toolshed

B.1. System Design

This work aims to enable learning and inference with multiple interactive vision tools for spatial reasoning. Effective tool-augmented spatial reasoning requires multi-turn, statedependent communication between the VLM and its vision tools. However, many essential tools such as object detectors, depth estimators, and 3D reconstruction modules, are computationally heavy and often dominate both inference and training time in VLM–RL pipelines. Moreover, modern training relies on batched generation, where multiple conversations are executed in parallel. In naïve implementations, a single blocking tool call can stall the entire batch, effectively reducing tool interaction to a serial process. This makes it crucial to keep tools continuously available, preloaded on device, and capable of serving multiple concurrent conversations. To address these challenges, we introduce **Toolshed** (visualized in Figure 1), a distributed toolkit that enables scalable, asynchronous, and parallel vision tool interaction.

- **Decoupled execution.** Tool invocations run independently from the policy's main inference loop, avoiding blocking calls that would otherwise stall unrelated computations.
- **Asynchronous Processing.** Multiple parallel instances can serve the same tool, each receiving inputs and producing outputs asynchronously, enabling high throughput even under large-scale rollouts.
- **Resources isolation.** Tool instances are assigned isolated resources based on the computational profile of each tool.
- **Environment isolation.** Each tool type is hosted in a dedicated python environment, solving the dependency compatibility issue that comes with hosting multiple computer vision tools in a single system.
- **Elastic scaling.** The system design supports automatic spawning of additional tool workers in response to bursts of tool usage, allowing throughput to remain stable even for large batch rollouts. (This capability is part of the infrastructure design but was not enabled in our training experiments.)
- **Multimodal data passing.** Seamless exchange of text, images, and structured variables (*e.g.*, 3D point clouds) is supported between the VLM and tools, even when they run on different devices or GPU nodes. This enables tool workflows that require different types of inputs and outputs.

In practice, Toolshed is implemented on top of the Ray[1] distributed execution framework, which provides lightweight task scheduling, actor management, and high-throughput message passing. For interactive reinforcement learning, Toolshed integrates seamlessly with VERL[2]: VERL's asynchronous multi-turn rollouts align naturally with Toolshed's asynchronous tool actors, enabling us to parallelize expensive perception, generation, and simulation steps without slowing down rollouts. This results in significantly higher steps-per-second compared to monolithic or synchronous training setups. For inference, Toolshed can be attached both to our trained model and to proprietary models (e.g., GPT-5, Claude) via simple API calls, enhancing their robotic spatial reasoning capabilities.

B.2. Provided Tools

Vision tools. We provide the following vision tools. `image_ops` offers basic image manipulations such as point- and mask-based indexing. `sam2` performs instance segmentation from one or more clicks, powered by Segment Anything 2 [4]. `point1` and `point2` are two object-pointing detectors backed by RoboRefer [9] and Molmo [3], respectively. `depth_estimator` predicts monocular depth and reconstructs 3D point clouds using DepthPro [1]. `compute_bbox` estimates 3D bounding boxes and object poses from reconstructed geometry, while `compute_grasp` predicts collision-free grasp poses for robotic manipulation. Finally, `code_executor` allows the VLM to execute small Python snippets for orchestrating multi-tool workflows, returning results with captured `stdout`/`stderr` and optionally caching intermediate outputs for reuse.

Robotic tools. We integrate a set of robotic tools that enable embodied perception and manipulation. `capture_image` captures RGB observations from the robot's onboard camera and stores them for subsequent visual processing. `capture_depth` acquire depth information from the scene, returning a depth map with focal length and a full 3D point cloud reconstruction. `execute_grasp` executes a grasp given a 4×4 transformation matrix representing the target grasp pose and reports execution success and timing feedback. `place_object` places an object at a specified 2D image coordinate, confirming successful placement in the returned text message. Apart from the above real robotic tools that control real-world robot arms, we also provide a set of `mock_robot` tools without relying on real robots for the ease of data generation and training. Together, these tools provide a physical interface that allows the VLM to perceive, reason, and act within the real world, enabling unified spatial reasoning and robotic control.

B.3. Example Process of Launching Tool IRL

We show a condensed workflow of launching tools and interactive reinforcement learning via a single bash script in Listing 1.

Listing 1. Example workflow for launching Toolshed and running VERL GRPO training with tool calling enabled.

```
# -------------------------------------
# Graceful cleanup (kills the Toolshed actor
    process so detached actors vanish)
# -------------------------------------
```

---

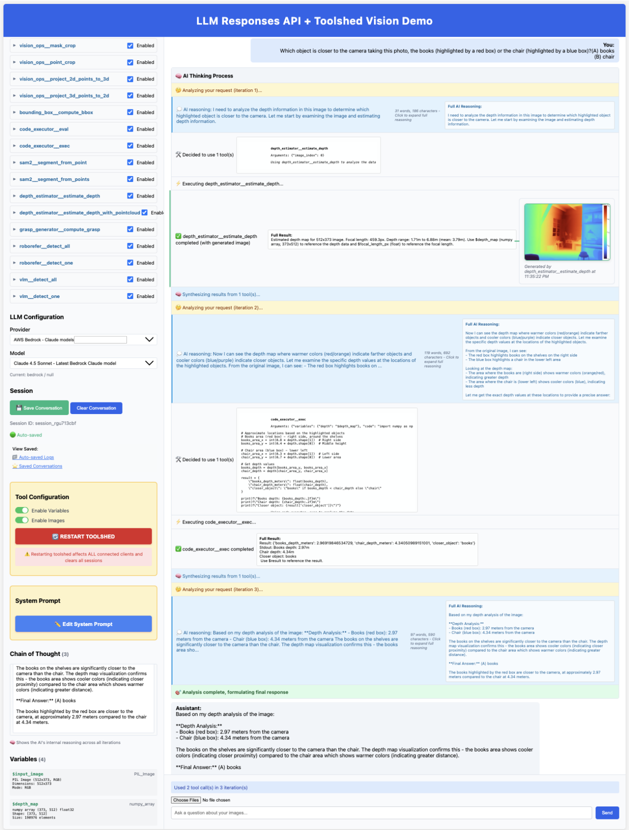
[1] ray.io/

[2] github.com/volcengine/verl

```
TOOLSHED_PID=""
cleanup() {
  echo "Cleaning_up"
  if [[ -n "$TOOLSHED_PID" ]]; then
    echo "Killing(PID=$TOOLSHED_PID)"
    kill $TOOLSHED_PID 2>/dev/null
    wait $TOOLSHED_PID 2>/dev/null
    echo "Stopped."
  fi
  exit 0
}
trap cleanup SIGINT SIGTERM EXIT

# 1. Launch Toolshed with GPU-backed vision
     tools
python - <<’PY’
import ray
from toolshed import start_toolkit

ray.init(address=’auto’)

tool_configs = {
    "point1": {"num_actors": 2, "resources": {"
         num_gpus": 0.5}},
    "point2": {"num_actors": 2, "resources": {"
         num_gpus": 0.5}},
    "sam2": {"num_actors": 4, "resources": {"
         num_gpus": 0.2}},
    "depth_estimator": {"num_actors": 4, "
         resources": {"num_gpus": 0.2}}
    ...
}

pg = ray.util.placement_group([{"CPU": 8, "GPU
     ": 8}], strategy="STRICT_PACK")
ray.get(pg.ready())
router = start_toolkit(tool_configs, dashboard
     =True, placement_group=pg)
print("Started:", list(tool_configs))
PY

# 2. Generate a YAML tool config and make sure
     tools are ready
python generate_toolshed_config.py --output
     toolshed_config.yaml

# 3. Launch VERL GRPO training with Toolshed
     Integration
python -m verl.trainer.main_ppo \
    actor_rollout_ref.model.path=Qwen/Qwen2.5-
         VL-3B-Instruct \
    ...
    actor_rollout_ref.rollout.multi_turn.
         tool_config_path=toolshed_config.yaml

echo "Finished."
```

B.4. Example Toolshed Integration with Proprietary Models

Figure 2 illustrates our integration of Toolshed with a proprietary model (Claude Sonnet 4.5), enabling seamless interactive communication among the user, external tools, and the VLM.



vision_ops__mask_crop ☑ Enabled
vision_ops__point_crop ☑ Enabled
vision_ops__project_2d_points_to_3d ☑ Enabled
vision_ops__project_3d_points_to_2d ☑ Enabled
bounding_box__compute_bbox ☑ Enabled
code_executor__eval ☑ Enabled
code_executor__exec ☑ Enabled
sam2__segment_from_point ☑ Enabled
sam2__segment_from_points ☑ Enabled
depth_estimator__estimate_depth ☑ Enabled
depth_estimator__estimate_depth_with_pointcloud ☑ Enabl
grasp_generator__compute_grasp ☑ Enabled
roborefer__detect_all ☑ Enabled
roborefer__detect_one ☑ Enabled
vlm__detect_all ☑ Enabled
vlm__detect_one ☑ Enabled

Session ID: session_rgu713cbf

Auto-saved

⚠ Restarting toolshed affects ALL connected clients and clears all sessions

The books on the shelves are significantly closer to the camera than the chair. The depth map visualization confirms this - the books area shows cooler colors (indicating closer proximity) compared to the chair area which shows warmer colors (indicating greater distance).

Final Answer: (A) books

The books highlighted by the red box are closer to the camera, at approximately 2.97 meters compared to the chair at 4.34 meters.

Shows the AI's internal reasoning across all iterations

```
$input_image                         PIL_Image
PIL Image (512x373, RGB)
Dimensions: 512x373
Mode: RGB

$depth_map                           numpy_array
numpy array (373, 512) float32
Shape: (373, 512)
Size: 190976 elements
```

You: Which object is closer to the camera taking this photo, the books (highlighted by a red box) or the chair (highlighted by a blue box)?(A) books (B) chair

Analyzing your request (iteration 1)...

AI reasoning: I need to analyze the depth information in this image to determine which highlighted object is closer to the camera. Let me start by examining the image and estimating depth information.

31 words, 186 characters - Click to expand full reasoning

Full AI Reasoning:

I need to analyze the depth information in this image to determine which highlighted object is closer to the camera. Let me start by examining the image and estimating depth information.

Decided to use 1 tool(s)

```
depth_estimator__estimate_depth
Arguments: {"image_index": 0}
Using depth_estimator__estimate_depth to analyze the data
```

Executing depth_estimator__estimate_depth...

✅ depth_estimator__estimate_depth completed (with generated image)

Full Result:
Estimated depth map for 512x373 image. Focal length: 459.3px. Depth range: 1.71m to 6.88m (mean: 3.79m). Use $depth_map (numpy array, 373x512) to reference the depth data and $focal_length_px (float) to reference the focal length.

Generated by depth_estimator__estimate_depth at 11:35:22 PM

Synthesizing results from 1 tool(s)...

Analyzing your request (iteration 2)...

AI reasoning: Now I can see the depth map where warmer colors (red/orange) indicate farther objects and cooler colors (blue/purple) indicate closer objects. Let me examine the specific depth values at the locations of the highlighted objects. From the original image, I can see: - The red box highlights books on ...

119 words, 692 characters - Click to expand full reasoning

Full AI Reasoning:

Now I can see the depth map where warmer colors (red/orange) indicate farther objects and cooler colors (blue/purple) indicate closer objects. Let me examine the specific depth values at the locations of the highlighted objects.

From the original image, I can see:
- The red box highlights books on the shelves on the right side
- The blue box highlights a chair in the lower left area

Looking at the depth map:
- The area where the books are (right side) shows warmer colors (orange/red), indicating greater depth
- The area where the chair is (lower left) shows cooler colors (blue), indicating less depth

Let me get the exact depth values at these locations to provide a precise answer:

Decided to use 1 tool(s)

```
code_executor__exec

Arguments: {"variables": {"depth": "$depth_map"}, "code": "import numpy as np

# Approximate locations based on the highlighted objects
# Books area (red box) - right side, around the shelves
books_area_x = int(0.8 * depth.shape[1])  # Right side
books_area_y = int(0.4 * depth.shape[0])  # Middle height

# Chair area (blue box) - lower left
chair_area_x = int(0.3 * depth.shape[1])  # Left side
chair_area_y = int(0.7 * depth.shape[0])  # Lower area

# Get depth values
books_depth = depth[books_area_y, books_area_x]
chair_depth = depth[chair_area_y, chair_area_x]

result = {
    \"books_depth_meters\": float(books_depth),
    \"chair_depth_meters\": float(chair_depth),
    \"closer_object\": \"books\" if books_depth < chair_depth else \"chair\"
}

print(f\"Books depth: {books_depth:.2f}m\")
print(f\"Chair depth: {chair_depth:.2f}m\")
print(f\"Closer object: {result['closer_object']}\")"}
```

Executing code_executor__exec...

✅ code_executor__exec completed

Full Result:
Result: {'books_depth_meters': 2.96919846534729, 'chair_depth_meters': 4.34050989151001, 'closer_object': 'books'}
Stdout: Books depth: 2.97m
Chair depth: 4.34m
Closer object: books
Use $result to reference the result.

Synthesizing results from 1 tool(s)...

Analyzing your request (iteration 3)...

AI reasoning: Based on my depth analysis of the image: **Depth Analysis:** - Books (red box): 2.97 meters from the camera - Chair (blue box): 4.34 meters from the camera The books on the shelves are significantly closer to the camera than the chair. The depth map visualization confirms this - the books area sho...

97 words, 590 characters - Click to expand full reasoning

Full AI Reasoning:

Based on my depth analysis of the image:

Depth Analysis:
- Books (red box): 2.97 meters from the camera
- Chair (blue box): 4.34 meters from the camera

The books on the shelves are significantly closer to the camera than the chair. The depth map visualization confirms this - the books area shows cooler colors (indicating closer proximity) compared to the chair area which shows warmer colors (indicating greater distance).

Final Answer: (A) books

The books highlighted by the red box are closer to the camera, at approximately 2.97 meters compared to the chair at 4.34 meters.

Analysis complete, formulating final response

Assistant:
Based on my depth analysis of the image:

Depth Analysis:
- Books (red box): 2.97 meters from the camera
- Chair (blue box): 4.34 meters from the camera

The books on the shelves are significantly closer to the camera than the chair. The depth map visualization confirms this - the books area shows cooler colors (indicating closer proximity) compared to the chair area which shows warmer colors (indicating greater distance).

Final Answer: (A) books

Used 2 tool call(s) in 3 iteration(s)

Figure 2. Interactive web demo illustrating Claude’s tool-augmented reasoning when integrated with Toolshed.

B.5. Detailed Tool APIs

Toolshed uses pythonic, modular interfaces & APIs (Appx B), *minimizing* new tool integration effort: only definition with `@tool_method` decorator is required; execution, RL integration, etc, are handled *automatically*. Complete API of the computer vision and robotic tools supported by Toolshed include:

`image_ops.point_crop(data, x, y)`

Purpose: Get the data value in the numpy ndarray ‘data‘ at the given coordinate.

- **Inputs**:
  - `data`: Numpy ndarray of shape (H, W) or (H, W, C), or PIL Image
  - `x`: Normalized x-coordinate, float in $[0, 1]$
  - `y`: Normalized y-coordinate, float in $[0, 1]$
- **Outputs**:
  - **Raw value**: `data` value indexed at the input coordinate.
  - **Text**: Reports the information about the pixel value at the given coordinates.

`image_ops.point_crop(image, points)`

Crop image to minimally encompass all given points.

- **Inputs**:
  - `image`: PIL.Image
  - `points`: list of (x, y) normalized floats in $[0, 1]$
- **Outputs**:
  - **Raw value**: `PIL.Image` cropped to the bounding box of the points
  - **Text**: Reports the crop box, size, and number of points
  - **Image**: The cropped region only (no overlay)
  - **Variables**: `cropped_image` (`PIL.Image`)

`image_ops.mask_crop(image, mask)`

Crop to the tight bounding box of a boolean mask; outside-mask pixels are set to white.

- **Inputs**:
  - `image`: PIL.Image
  - `mask`: boolean `numpy.ndarray` of shape $H \times W$ matching the image
- **Outputs**:
  - **Raw value**: `PIL.Image` of the masked region on white background, cropped to mask bounds
  - **Text**: Reports crop box, size, and mask coverage percentage
  - **Image**: Masked crop on white background (no overlay)
  - **Variables**: `masked_crop` (`PIL.Image`)

`sam2.segment_from_point(image, x, y)`

Segment the object at a single pixel coordinate.

- **Inputs**:
  - `image`: PIL.Image
  - `x`: Normalized x-coordinate, float in $[0, 1]$
  - `y`: Normalized y-coordinate, float in $[0, 1]$
- **Outputs**:
  - **Raw value**: `dict` with `mask` (boolean $H \times W$ `numpy.ndarray`) and `iou_score` (float)
  - **Text**: Reports the click location and IoU score
  - **Image**: Original image with semi-transparent green mask, a white mask outline, and a red circular point marker (white outline)
  - **Variables**: `segmentation_mask` (boolean $H \times W$ `numpy.ndarray`)

`sam2.segment_from_points(image, points)`

Segment an object using multiple foreground points.

- **Inputs**:
  - `image`: PIL.Image
  - `points`: list of (x, y) normalized floats in $[0, 1]$
- **Outputs**:
  - **Raw value**: `dict` with `mask` (boolean $H \times W$ `numpy.ndarray`) and `iou_scores` (1-D `numpy.ndarray`)
  - **Text**: Reports the number of points and the best IoU score
  - **Image**: Original image with semi-transparent green mask, a white mask outline, and red circular markers at all provided points.
  - **Variables**: `segmentation_mask` (boolean $H \times W$ `numpy.ndarray`)

`point1.detect_one(image, obj_name)`

Identify one instance of the named object by pointing to them with Roborefer.

- **Inputs**:
  - `image`: PIL.Image
  - `obj_name`: string
- **Outputs**:
  - **Raw value**: String serialization of a normalized point coordinate $(x, y) \in [0, 1]^2$
  - **Text**: Reports the object name, count, and the normalized point
  - **Image**: Original image with red circular point markers at detected locations (white outlines)
  - **Variables**: `<obj_name>_detection` ((x, y) floats in $[0, 1]$; spaces in *obj_name* replaced with underscores)

`point1.detect_all(image, obj_name)`

Identify instances of the named object by pointing to them with Roborefer.

- **Inputs**:
  - `image`: PIL.Image
  - `obj_name`: string
- **Outputs**:
  - **Raw value**: String serialization of a list of normalized point coordinates $(x, y) \in [0, 1]^2$
  - **Text**: Reports the object name, count, and the list of normalized points
  - **Image**: Original image with red circular point markers at detected locations (white outlines)
  - **Variables**: `<obj_name>_detections` (list of (x, y) floats in $[0, 1]$; spaces in *obj_name* replaced with underscores)

`point2.detect_one(image, obj_name)`

Identify one instance of the named object by pointing to them with Molmo.

- **Inputs**:
  - `image`: PIL.Image
  - `obj_name`: string
- **Outputs**:
  - **Raw value**: String serialization of a normalized point coordinate $(x, y) \in [0, 1]^2$
  - **Text**: Reports the object name, count, and the normalized point

  - **Image**: Original image with red circular point markers at detected locations (white outlines)
  - **Variables**: `<obj_name>_detection` ((x, y) floats in $[0, 1]$; spaces in *obj_name* replaced with underscores)

`point2.detect_all(image, obj_name)`

Identify instances of the named object by pointing to them with Molmo.

- **Inputs**:
  - `image`: PIL.Image
  - `obj_name`: string
- **Outputs**:
  - **Raw value**: String serialization of a list of normalized point coordinates $(x, y) \in [0, 1]^2$
  - **Text**: Reports the object name, count, and the list of normalized points
  - **Image**: Original image with red circular point markers at detected locations (white outlines)
  - **Variables**: `<obj_name>_detections` (list of (x, y) floats in $[0, 1]$; spaces in *obj_name* replaced with underscores)

`depth_estimator.estimate_depth(image)`

Monocular depth estimation with DepthPro.

- **Inputs**:
  - `image`: PIL.Image
- **Outputs**:
  - **Raw value**: `dict` with `depth_map` ($H\times W$ float array, meters), `focal_length_px` (float), `width` (int), `height` (int)
  - **Text**: Reports image size, focal length, and depth range statistics
  - **Image**: Colorized depth map with a vertical scale bar on the right labeled "Depth (m)"
  - **Variables**: `depth_map` ($H\times W$ float array), `focal_length_px` (float)

`depth_estimator.estimate_depth_with_pointcloud(image)`

Monocular depth estimation and 3D point cloud generation with DepthPro.

- **Inputs**:
  - `image`: PIL.Image
- **Outputs**:
  - **Raw value**: `dict` with `depth_map` ($H\times W$ float array, meters), `point_cloud` ($N\times 3$ float array of 3D points in camera coordinates), `focal_length_px` (float), `width` (int), `height` (int)
  - **Text**: Reports image size, focal length, and depth range statistics
  - **Image**: Colorized depth map with a vertical scale bar on the right labeled "Depth (m)"
  - **Variables**: `depth_map` ($H\times W$ float array), `point_cloud` ($N\times 3$ float array of 3D points in camera coordinates), `focal_length_px` (float)

`grasp_generator.compute_grasp(point_cloud, mask, image, focal_length_px)`

Generate a single grasp pose for a masked subset of a point cloud with GraspGen.

- **Inputs**:
  - `point_cloud`: $N\times 3$ numpy float array
  - `mask`: Boolean segmentation mask
  - `image`: PIL.Image
  - `focal_length_px`: float
- **Outputs**:
  - **Raw value**: Collision-free grasp pose, and collision-free confidence
  - **Text**: Reports the collision-free grasp confidence, the total number of generated grasps and the percentage of collision-free grasps, and the projected 2D gripper points of the best grasp pose in normalized coordinates
  - **Image**: Original image overlayed with projected X-(red), Y-(green), Z-(blue) gripper axes
  - **Variables**: `grasp_pose` (4×4 ndarray, OpenCV camera frame)

`3d_bbox.compute_bbox(point_cloud, mask, focal_length_px)`

Compute an oriented bounding box for a masked subset of a point cloud.

- **Inputs**:
  - `point_cloud`: $N\times 3$ numpy float array
  - `mask`: Boolean segmentation mask
  - `focal_length_px`: float
- **Outputs**:
  - **Raw value**: Box corners in 3D, box corners in 2D, edges, and extent
  - **Text**: Summary containing number of input points, the point coordinates in 3d and 2d, mask shape, box extents, and edges
  - **Image**: No image output
  - **Variables**: `obb_corners_3d` (8×3 list of lists, meters in opencv camera frame), `obb_corners_2d` (8×2 list of lists, normalized image coordinates), `extent` (3-element ndarray, extent of the bounding box in meters), `edges` (list of pairs of integers, edges of the bounding box defined by the indices of the corners)

`code_executor.exec(code)`

Execute a multi-line Python block (imports limited to `math` and `numpy`).

- **Inputs**:

  - `code`: string
- **Outputs**:
  - **Raw value**: tuple $(result, stdout, stderr)$
  - **Text**: Summarizes the result, captured stdout, and stderr; notes a stored variable if applicable
  - **Image**: No image output
  - **Variables**: `last_exec_result` (present iff a non-`None` result and variables are enabled)

`code_executor.eval(expression)`
Evaluate a single Python expression.

- **Inputs**:
  - `expression`: string
- **Outputs**:
  - **Raw value**: tuple $(result, stdout, stderr)$
  - **Text**: Summarizes the result, captured stdout, and stderr; notes a stored variable if applicable
  - **Image**: No image output
  - **Variables**: `last_eval_result` (present iff a non-`None` result and variables are enabled)

`mock_robot.capture_image(mock_data)`
Return the mock image from the dataset without a real robot.

- **Inputs**:
  - `mock_data`: `dict` with `mock_image`
- **Outputs**:
  - **Raw value**: Image from mock camera
  - **Text**: Image dimensions and capture status
  - **Image**: Captured image from mock camera
  - **Variables**: `captured_image` (PIL Image)

`mock_robot.get_depth(mock_data)`
Return the mock depth from the dataset without a real robot.

- **Inputs**:
  - `mock_data`: `dict` with `mock_image` (PIL Image), `mock_depth_map` (numpy array), `mock_focal_length_px` (float), `image_width` (int), `image_height` (int)
- **Outputs**:
  - **Raw value**: `image` (PIL Image), `depth_map` (numpy array), `focal_length_px` (float), `width` (int), `height` (int)
  - **Text**: Summary of depth data including image dimensions, focal length, and depth statistics
  - **Image**: A colorized depth map visualization where closer objects appear cooler (blue/purple) and distant objects appear warmer (red/yellow)
  - **Variables**: `depth_map` (2D numpy array of depth values in meters), `foca_length_px` (float, estimated focal length in pixels)

`mock_robot.`
`get_depth_with_pointcloud(mock_data)`
Return the mock depth and and point cloud generation from the dataset without a real robot.

- **Inputs**:
  - `mock_data`: `dict` with `mock_image` (PIL Image), `mock_depth_map` (numpy array), `mock_point_cloud` (numpy array), `mock_focal_length_px` (float), `image_width` (int), `image_height` (int).
- **Outputs**:
  - **Raw value**: `image` (PIL Image), `depth_map` (numpy array), `mock_point_cloud` (numpy array), `focal_length_px` (float), `width` (int), `height` (int).
  - **Text**: Summary of depth data and and point cloud generation including image dimensions, focal length, depth statistics, and point cloud size
  - **Image**: A colorized depth map visualization where closer objects appear cooler (blue/purple) and distant objects appear warmer (red/yellow)
  - **Variables**: `depth_map` (2D numpy array of depth values in meters), `point_cloud` ($N \times 3$ array of 3D points), `foca_length_px` (float, estimated focal length in pixels)

`mock_robot.execute_grasp(grasp_pose)`
Simulate executing a grasp (always succeeds).

- **Inputs**:
  - `grasp_pose`: 4×4 transformation matrix representing the grasp pose in the robot's camera frame, OpenCV convention
- **Outputs**:
  - **Raw value**: `success` (boolean), `execution_time_s` (float)
  - **Text**: Confirmation that grasp was successful
  - **Image**: No image output
  - **Variables**: No variable output

`mock_robot.`
`place_object_at_2d_location(point_2d)`
Simulate placing object at 2D location (always succeeds).

- **Inputs**:
  - `point_2d`: 2D normalized image coordinate where the object should be placed
- **Outputs**:
  - **Raw value**: `success` (boolean), `execution_time_s` (float)
  - **Text**: Confirmation that placement was successful
  - **Image**: No image output
  - **Variables**: No variable output

`mock_robot.place_object_at_3d_location(point_3d)`
Simulate placing object at 3D location (always succeeds).

- **Inputs**:
  - `point_3d`: 3D point in the robot's camera frame (list or numpy array) where the object should be placed
- **Outputs**:
  - **Raw value**: `success` (boolean), `execution_time_s` (float)
  - **Text**: Confirmation that placement was successful
  - **Image**: No image output
  - **Variables**: No variable output

`robot.capture_image()`
Return the mock image from the dataset without a real robot.

- **Inputs**: No input required
- **Outputs**:
  - **Raw value**: `image_shape` (numpy array or list), `image` (PIL Image)
  - **Text**: Image dimensions and capture status
  - **Image**: Captured image from robot camera
  - **Variables**: `captured_image` (PIL Image)

`robot.get_depth()`
Retrieve depth map from the robot's depth sensor.

- **Inputs**: No input required
- **Outputs**:
  - **Raw value**: `depth_map` (numpy array), `depth_map_visualization` (PIL Image), `focal_length_px` (float), `width` (int), `height` (int)
  - **Text**: Summary of depth data including image dimensions, focal length, and depth statistics
  - **Image**: A colorized depth map visualization where closer objects appear cooler (blue/purple) and distant objects appear warmer (red/yellow)
  - **Variables**: `depth_map` (2D numpy array of depth values in meters), `foca_length_px` (float, estimated focal length in pixels)

`robot.get_depth_with_pointcloud()`
Retrieve depth map from robot's depth sensor and generate 3D point cloud.

- **Inputs**: No input required
- **Outputs**:
  - **Raw value**: `image` (PIL Image), `depth_map` (numpy array), `point_cloud` (numpy array), `focal_length_px` (float), `width` (int), `height` (int).
  - **Text**: Summary of depth data and and point cloud generation including image dimensions, focal length, depth statistics, and point cloud size
  - **Image**: A colorized depth map visualization where closer objects appear cooler (blue/purple) and distant objects appear warmer (red/yellow)
  - **Variables**: `depth_map` (2D numpy array of depth values in meters), `point_cloud` ($N \times 3$ array of 3D points), `foca_length_px` (float, estimated focal length in pixels)

`robot.execute_grasp(grasp_pose)`
Execute a grasp by moving the robot to the specified pose via a pre-grasp point, and closing the gripper.

- **Inputs**:
  - `grasp_pose`: 4×4 transformation matrix representing the grasp pose in the robot's camera frame, OpenCV convention
- **Outputs**:
  - **Raw value**: `success` (boolean), `execution_time_s` (float), `image` (PIL Image)
  - **Text**: Status of the grasp execution
  - **Image**: View from robot camera after the grasp is executed
  - **Variables**: `captured_image` after the grasp is executed

`robot.place_object_at_2d_location(point_2d)`
Move the robot to a place it's currently held object based on a 2D normalized image coordinate. The tool will convert to a 3D placement location automatically by shooting a ray.

- **Inputs**:
  - `point_2d`: 2D normalized image coordinate where the object should be placed
- **Outputs**:
  - **Raw value**: `success` (boolean), `execution_time_s` (float), `image` (PIL Image)
  - **Text**: Status of the release operation
  - **Image**: View from robot camera after the placement is executed
  - **Variables**: `captured_image` (PIL Image) after the placement is executed

`robot.place_object_at_3d_location(point_3d)`
Move the robot to a 3D placement point and open the gripper to place the object.

- **Inputs**:

  - `point_3d`: 3D point in the robot's camera frame (list or numpy array) where the object should be placed
- **Outputs**:
  - **Raw value**: `success` (boolean), `execution_time_s` (float), `image` (PIL Image)
  - **Text**: Status of the placement operation
  - **Image**: View from robot camera after the placement is executed
  - **Variables**: `captured_image` (PIL Image) after the placement is executed

C. Additional Method Details

C.1. Group Relative Policy Optimization

We employ Group Relative Policy Optimization (GRPO) [5] as our RL training algorithm. We present the details of GRPO below.

For each input $\mathcal{I}$, in total N rollout procedures are launched asynchronously under the current policy π_θ. Each rollout generates in total N multi-turn rollouts $O_1, O_2, \ldots, O_N$. Their rewards are calculated as $r_1, r_2, \ldots, r_N$. Each r_i is standardized into a relative advantage A_i via group computation:

$$A_i = \frac{r_i - \text{mean}(\{r_1, r_2, \ldots, r_N\})}{\text{std}(\{r_1, r_2, \ldots, r_N\})}. \tag{1}$$

The policy is then optimized by minimizing the GRPO loss:

$$\begin{aligned} \mathcal{L}_{\text{GRPO}}(\theta) = \mathbb{E}_i \Big[& - \min\big(\rho_i A_i,\, \text{clip}(\rho_i, 1{-}\epsilon, 1{+}\epsilon) A_i\big) \\ & + \beta\, \text{KL}\big(\pi_\theta \,\|\, \pi_{\text{ref}}\big) \Big], \end{aligned} \tag{2}$$

where $\rho_i = \frac{\pi_\theta(i)}{\pi_{\text{ref}}(i)}$, and π_{ref} denotes the reference policy model, i.e., the VLM trained after stage 1. Here, ϵ and β are tunable hyperparameters controlling the clipping range and KL regularization strength. This formulation encourages the policy to increase the probability of high-reward responses while maintaining stability through KL regularization.

C.2. Alternative Reward Design

Other reward for the pointing questions. Considering that pointing is the first step of solving many spatial reasoning tasks or using other tools, we have experimented with several different rewards for pointing before finalizing the NNDC reward. We show results in Appendix E.4, emphasizing the importance the reward design for tasks requiring explicit numerical estimation.

System Prompt

You are an expert in 3D spatial reasoning for robotics.
Given a spatial reasoning task, **follow this process**:
1. First, think about the reasoning process as an internal monologue the first time you receive the question, and every time you receive new information. Your reasoning process MUST be enclosed within <think> </think> tags.
2. After thinking, if you need additional information to answer the question, or conduct external control, call the appropriate tool.
3. When you receive a tool response, use that information to continue your analysis.
4. Once no further visual analysis or tool calls are needed, you **MUST** provide your final answer inside <answer> </answer> tags without detailed illustrations.
Example answer format: <answer> <Your final answer here> </answer>
Tools
You may call one or more functions to assist with the user query.
You are provided with function signatures within <tools> </tools> tags:
<tools> {tools} </tools>
For each function call, return a **JSON** object with function name and arguments within <tool_call> </tool_call> tags:
<tool_call> {"name": <func-name>, "arguments": <args-json-object>} </tool_call>

Figure 3. **System prompt.** Instructional prompt guiding the model's reasoning, tool-call, and answer process.

- **Binary:**

$$R_{\text{B}} = \begin{cases} 1.0, & \text{if the predicted point lies within} \\ & \text{the ground truth convex hull,} \\ 0, & \text{otherwise.} \end{cases}$$

- **Normalized Signed Distance to Hull (NSDH):**

$$R_{\text{NSDH}} = \begin{cases} 0.5 + 0.5\exp(s), & \text{if } s \le 0, \\ 0.5\exp(-s), & \text{if } s > 0, \end{cases}$$

  where s is the signed distance from the predicted point to the convex hull boundary (negative inside, positive outside).
- **Normalized Area Change (NAC):** Let ΔA be the change in area after adding the predicted point to the convex hull, and A_0 be the original area. Then:

$$R_{\text{NAC}} = \exp\left(-\frac{\Delta A}{A_0}\right).$$

Similar to NNDC, by default, we also apply clipping with the binary accuracy term to emphasize precision for alternative non-binary rewards. (*e.g.*, $R = \max(R_{\text{NSDH}}, R_{\text{B}})$)

Format score details. In addition to task-specific rewards, we explored defining a *format score* to enforce the structural correctness of model outputs (as defined in the system prompt shown in Figure 3), but did not use it in the final training. The format score verifies that every `<tool_call>` tag is immediately preceded by a `<think>` tag, and that the single final `<answer>` is also directly preceded by a `<think>`. The output must contain exactly one `<answer>` tag, positioned at the end of the response. Additional optional constraints can be applied, such as requiring at least one `<tool_call>` in the output.

Predictions failing to meet these structural requirements receive a format score of zero, while perfectly formatted predictions receive a score of one.

The final reward is computed as a weighted sum of the accuracy-based reward and the format score:

$$R_{\text{final}} = R_{\text{acc}} + \lambda\, R_{\text{format}},$$

where R_{acc} denotes the task-specific accuracy reward, $R_{\text{format}} \in \{0, 1\}$ is the format score (equal to 1.0 if all structural criteria described above are satisfied, and 0 otherwise), and $\lambda \in [0, 1]$ is a scalar weight controlling the influence of the format score. Following prior work, λ is often set to 0.3.

D. Additional Implementation Details

D.1. More Training and Compute Details

Prompts and teaching data. For Phase-1 SFT of the base model and Phase-2 IRL, we use the system prompt shown in Figure 3. To improve the effectiveness of Phase-1 IRL training for the teacher model, we augment this prompt with two tool-use examples: one demonstrating how to solve a spatial relationship problem using the pointing tool, and another illustrating 2D bounding box estimation with the same tool. In addition, when generating teaching data with Claude Sonnet 4.5, we include extra instructional prompts that encourage careful interpretation of tool outputs and better understanding of image coordinate systems.

Synthetic robot data for teaching. Due to the substantial latency of robot-in-the-loop training and data collection, we construct grasp and place data using the HOPE dataset and provide it to Claude Sonnet 4.5 together with mock robot tools to generate the robotic portion of the teaching dataset. The interactive learning stages themselves do not use this synthetic robot-tool component; instead, they focus exclusively on spatial reasoning with vision tools.

Answer balancing. Another practical consideration is maintaining balanced answer distributions for multiple-choice questions. For example, the original RoboSpatial VQA dataset contains more than 75% "no" answers, which biases the VLM toward predicting "no" during both SFT and IRL. We find that rebalancing the data mitigates this issue and improves answer calibration across tasks.

Hyperparameters and training stability. The major hyperparameters used during training are summarized in Table 1. Across our experiments, the KL coefficient emerges as a particularly important hyperparameter: a relatively small value is needed to encourage sufficient exploration during RL. However, with older versions of Verl, this choice can introduce training instability. In particular, during Phase-1 IRL, we observe an initial reward drop when the KL coefficient is too small. We explored several mitigation strategies, including format rewards, format penalties, alternative KL loss formulations, and related variants, but none fully eliminated this effect. Interestingly, the instability is alleviated when integrating Toolshed with the new agent loop feature introduced in newer versions of Verl. For this reason, we open-source our code based on integration with the latest Verl fork.

Table 1. Training configurations for Phase-1 IRL, Phase-1 SFT, and Phase-2 IRL. A dash (–) indicates that the setting is not applicable to that phase.

	Phase-1 IRL	**Phase-1 SFT**	**Phase-2 IRL**
Data			
Dataset	Direct VQA	Teaching tool-use	Direct VQA
#Samples	4k	8k	≈8k
Model			
Trainable Part	Language model (LLM) only; vision encoder + projector frozen		
#Tunable Parameters	2.55B		
Training			
Batch Size	64	8	64
Learning Rate	1e-6	1e-5	1e-6
Epoch	5	2	2
Warmup Ratio	0.0	0.1	0.0
LR Schedule	NA	cosine	NA
KL Coefficient	1e-4	–	1e-4
Entropy Coefficient	0.0	–	0.0
Temperature	1.0	–	1.0
Max Prompt Length	8192	8192	8192
Max Response Length	8192	8192	8192
Rollout Number	5	–	5
#GPU (VLM)	8	8	8
#GPU (Tools)	8	–	8

D.2. MACE Metric for Grasp Affordances

The grasp estimation task requires the model to predict five key points in normalized image coordinates: the grasp center, left finger base, right finger base, left finger tip, and right finger tip. From these points, we define four finger direction vectors: grasp center → left finger base, grasp center → right finger base, left finger base → left finger tip, and right finger base → right finger tip.

The Mean Angular Coordinate Error (MACE) metric is defined as follows. Given the predicted and ground-truth grasp centers $\hat{c}$ and c, and the set of four corresponding finger direction vectors $\{\hat{r}_k\}_{k=1}^4$ and $\{r_k\}_{k=1}^4$, we define:

$$\text{MACE} = 1 - \frac{1}{2}\left(\frac{\|\hat{c} - c\|_2}{w} + \frac{1}{4}\sum_{k=1}^{4}\frac{1 - \cos(\hat{r}_k, r_k)}{2}\right), \tag{3}$$

where w denotes the gripper width used for spatial normalization, and $\cos(\hat{r}_k, r_k) = \frac{\hat{r}_k \cdot r_k}{\|\hat{r}_k\|\|r_k\|}$ represents the cosine similarity between the predicted and ground-truth orientations of the k-th finger-related vector.

Task	Ours	Claude	GPT
Pick			
Pick up the dark blue object	✓	✓	✓
Pick up the soft toy	×	×	×
Pick up the solid toy	✓	✓	×
Pick up the tall cylindrical tennis ball container	✓	✓	✓
Pick up the coconut water	✓	✓	✓
Pick up the plastic bottle	✓	✓	✓
Pick up the red box	✓	✓	✓
Relational Pick			
Pick up the far coconut water	✓	×	×
Pick up the coconut water that is closer to the camera	✓	✓	×
Pick up the left pineapple juice can	✓	×	×
Pick up the right pineapple juice can	✓	✓	×
Pick up the further purple drink	×	✓	✓
Pick up the near purple bottle	✓	×	✓
Pick & Place			
Pick up the hot dog and place it in the black bin	0	2	2
Pick up the tall cylindrical container and place it in the transparent bin	2	2	2
Pick up the leftmost condiment and place it in the transparent bin	2	2	2
Pick up the cinnamon and place it in front of the rice box	2	2	0
Pick up the rice box and place it next to the hot sauce	2	1	0
Pick up the plushie and place it left of the coconut water	2	1	1
Pick up the pony and place it left of the two plushies	2	1	2

Table 2. Per-task breakdown of the real-world manipulation results, comparing **Ours** (SpaceTools), Claude Sonnet 4.5 and GPT-5.

D.3. Robot Manipulation Setup

Robot System We conduct robot experiments on a Kinova Jaco arm equipped with the CuRobo [6] motion planner and a ZED2 RGB-D camera. We expose the robot as a tool and make it available to the VLM. The tool has functions: `capture_image` retrieves the current RGB image from the camera, `get_depth` and `get_depth_with_pointcloud` retrieve the current depth image, optionally with a pointcloud in the robot frame, `execute_grasp` moves the end-effector to a specified grasp pose via a pre-grasp point and closes the gripper, `place_object_at_2d_location` and `place_object_at_3d_location` offer two ways to parameterize a place operation that moves the robot hand holding an object over a location in the scene and opens the gripper. All motions are executed with the motion planner.

Robot Experiments Tasks and Results We design a suite of tasks across three categories. *Pick*, *Relational Pick*, and *Pick & Place*. We score both *Pick* tasks based on the success rate, and *Pick & Place* based on partial success rate, awarding 1 point each for a correct pick and place operation.

The full results at individual task level are in Table 2, omitting methods that fail to achieve any points. In *Pick up the soft toy* task, all models failed due to a common failure in pointing tool not being able to differentiate the soft toy from a rigid toy. In *Relational Pick* and *Pick & Place* tasks, SpaceTools shows superior ability than Claude and GPT-5 in correctly using the pointing tool to resolve spatial relations, reflecting an understanding of its strengths and limitations likely attributable to the interactive training with the tool.

Additional Details on Robot SFT Data Collection In order to collect SFT data of from the Universal Teacher (Claude Sonnet 4.5) on using the robot tool, we design a "mock robot" that has the same API as the robot tool, but it always simulates successful actions provided the API was called with valid arguments. This allows collecting a small number ($\sim$500) examples of valid robot API calls without requiring the physical robot in the loop and ensuring that our robot is unseen during training.

D.4. Details of Non-interactive RL Baseline

We present the detailed description of the Tool NIRL baseline referenced in the ablation section of the main paper, as space limitations prevented us from including all details there.

We follow the conventional tool-learning setup used in LLMs [8] to perform reinforcement learning of tool usage *without* executing tools during training. The core idea is to compute the reward solely from the correctness of the predicted tool name and its arguments, which requires access to ground-truth tool call traces for supervision. After obtaining tool-augmented reasoning traces from Claude, each multi-turn trace with T turns is decomposed into T single-turn training instances: the ith instance contains the conversation history up to turn i as input, and the corresponding ground-truth output for turn i as the target.

During training, for tool-call turns, we adopt the binary reward used in [8]. A reward of 1.0 is given only when both the tool call format and the tool call content match the ground truth:

$$r = \begin{cases} 1, & \text{if } \texttt{FormatCorrect} \land \texttt{ToolCallMatch} \\ 0, & \text{otherwise}, \end{cases} \tag{4}$$

where `FormatCorrect` verifies that the model output is wrapped in the required tags, and `ToolCallMatch` checks that both the tool name and its arguments exactly match the ground-truth tool call. For final-answer turns (i.e., non-tool turns), we reuse the same task-specific normalized rewards introduced in this paper.

E. Additional Experimental Results

E.1. Efficiency of Toolshed system.

A naive integration of computer vision tools in Verl is not feasible, since Verl instantiates tools when calling them, and

Table 3. Benefit of scaling tool instances with Toolshed under contention. We measure 8 simultaneous RoboRefer tool calls. Compared with a naive HTTP-based deployment using a single instance, Toolshed with 3 instances substantially reduces end-to-end latency.

Metric	**Naive HTTP (1 instance)**	**Toolshed (3 instances)**
Wall-clock Time ↓	8.5 ± 0.3 s	$\mathbf{2.7 \pm 0.1}$ s
Speedup ↑	baseline	$\mathbf{3.2}\times$

Table 4. Pipeline execution latency for answering “Is bok choy or clock closer?” using 2× RoboRefer, 2× SAM, 1× depth estimation, and 2× `index_at` tools. Toolshed improves both wall-clock latency and throughput, with further gains when scaling to 3 tool instances.

Metric	**HTTP (1 inst)**	**Toolshed (1 inst)**	**Toolshed (3 inst)**
Wall-clock Time ↓	20.23 ± 0.42 s	15.13 ± 0.08 s	$\mathbf{10.56 \pm 0.01}$ s
Throughput ↑	0.40 pipe/sec	0.53 pipe/sec	$\mathbf{0.76}$ pipe/sec
Per-Pipeline Time ↓	15,678 ms	11,998 ms	$\mathbf{7,931}$ ms

Table 5. I/O time breakdown (ms) by tool when no queueing is required. Toolshed is especially advantageous for tools with larger inputs/outputs, such as SAM and depth estimation, while maintaining low overhead across all steps.

Step	**HTTP I/O**	**Toolshed I/O**
`roborefer_bok_choy`	6 ± 0	10 ± 1
`roborefer_clock`	7 ± 0	10 ± 0
`sam2_bok_choy`	18 ± 2	11 ± 0
`sam2_clock`	18 ± 1	11 ± 0
`depth_estimation`	45 ± 6	11 ± 0
`index_*`	0	9 ± 0

parameter loading can take 5-10 minutes. The most compatible integration might be serving each tool with an HTTP server. We ran an experiment executing a pipeline of 2x pointing, 2x SAM, 2x depth, and 2x indexing tools, with overall results summarized in Table 3, Table 4, and Table 5. The latency of Toolshed is better for large inputs/outputs, like we encounter for computer vision tools. Toolshed also provides more efficient queuing for tools when 8 callers are contending for 1 or 3 available tools. Other than performance, toolshed has additional advantages like seamless development, allowing to write pythonic tools (inputs/outputs can be any python objects, including gpu tensors), scaling to multiple tool instances with a single config change, and python environment isolation.

E.2. A Closer Look at Tool IRL Alone.

While direct IRL over diverse tools poses challenges due to the vast action space, we demonstrate its effectiveness within a constrained setup using the RoboSpatial dataset and pointing tools. As shown in Table 6, this approach substantially improves spatial reasoning compared with both

Table 6. Comparison of Qwen2.5-VL-3B, its inference variants, and fine-tuned models on RoboSpatial and RefSpatial. *Direct Inference* refers to answering the question without intermediate reasoning or tool use. *CoT* denotes chain-of-thought inference. *+Toolshed* indicates tool-augmented inference without any additional training. Among all variants, Tool IRL achieves the highest overall accuracy on RoboSpatial and the strongest generalization to RefSpatial.

Model	**RoboSpatial**			**RefSpatial**
	VQA	**Vacant**	**Overall**	
Inference Baseline (no fine-tuning)				
Direct Inference	53.07	0.00	35.71	0.00
CoT	66.67	0.00	43.71	0.00
+Toolshed	47.37	9.02	34.00	17.69
Fine-tuned on RoboSpatial				
Tool-free SFT	75.88	13.11	54.00	0.00
Tool-free RL	72.37	20.49	54.28	0.00
Tool IRL	77.64	62.30	72.30	34.30

direct tool-free SFT and vanilla tool-free GRPO baselines (e.g., classic reasoning models like R1 [2]), as well as other inference approaches [7]. On RoboSpatial, the IRL with Tools model achieves 72.3% overall accuracy, outperforming SFT and vanilla GRPO. Notably, IRL with Tools is the only method that generalizes to unseen tasks: achieving 34.3% on RefSpatial, whereas other fine-tuning strategies yield zero accuracy. These results show that interactive tool use during RL enables the model to internalize transferable geometric reasoning skills beyond the training domain. Even without fine-tuning, connecting Toolshed to the pretrained model yields measurable improvements on RefSpatial, highlighting the intrinsic generalization benefit of tool-augmented spatial reasoning.

E.3. Scaling to New Tools with DIRL

DIRL supports learning a more diverse and adaptive set of tools than prior work, while also enabling new tools to be added without fully retraining the entire system. Even in the worst case, only partial retraining is needed. In particular, when introducing a new tool, the interactive RL stage for the foundational pointing tool—which accounts for roughly 60% of the total training compute—can always be reused. As a result, the worst-case cost of incorporating a new tool is only about 40% of the compute required for full DIRL training. See results in Table 7.

We further demonstrate there a more efficient strategy for new-tool adaptation. Starting from a DIRL model trained without the grasp prediction tool, we extend it using only supervised fine-tuning (SFT) and the final IRL stage on grasp data. This lightweight recipe requires only about 10% of the full training compute while achieving comparable performance. As a representative case study, we evaluate

Table 7. Adding a new tool to DIRL using grasp prediction as a representative example. Even in the worst case, DIRL only requires partial retraining, since the interactive RL stage for the foundational pointing tool can be reused. We also show an efficient adaptation recipe that uses only SFT and the final IRL stage on grasp data.

Setting	RoboSpatial	Grasp-SR	Compute (% of full DIRL)
Worst case	70%	50%	∼40%
Efficient example	62%	50%	∼10%

the addition of a grasp prediction tool. A broader investigation of continual learning with newly introduced tools and tasks remains an important direction for future work.

E.4. Other Ablations

Direct IRL on all tasks with all tools. As mentioned in the main paper, directly reinforcement learning with all tools on all tasks result in a large search space and is hard to learn effectively. We provide qualitative performance in Table 8, supporting this argument.

Table 8. Direct IRL on all tasks (*Direct IRL All.*) with all tools compared with our method.

Variant	IRL-T	Univ-T	S2-IRL	RoboSpatial	RefSpatial	Pose	Mean
SpaceTools (Ours)	✓	✓	✓	70.00	53.07	34.37	52.48
Direct IRL All.	×	×	✓	52.86	3.25	3.26	19.79

Reward and prompt for IRL. Due to computational constraints that prevent running full-scale IRL ablations, we evaluate different pointing rewards (*e.g.*, NSDH, NAC, Binary) and prompt design choices introduced in Appendix C.2 on a subset of 1k vacant-space localization questions from RoboSpatial using the Molmo pointing tool. The results in Table 9 show that NNDC without an additional format reward yields the most stable and reliable learning behavior. Accordingly, we adopt NNDC (without a format reward) for all subsequent training stages. These experiments also highlight the importance of normalizing rewards to the $[0, 1]$ range, a practice we apply consistently across all tasks. More broadly, this study underscores the richness of the reward-design space for spatial reasoning tasks, especially those requiring explicit numerical estimation.

Table 9. Ablation on reward and prompt design for the pointing task as introduced in Appendix C.2. *Norm.* indicates whether normalization to range $[0, 1]$ is applied to the reward function. *Clip.* indicates whether binary clipping is applied. *Format* indicates whether the format reward is applied. *Example in Prompt* indicates whether two tool-use examples are added in the prompt. Checkmarks indicate which components are included for each variant.

Reward Variant	Norm.	Clip.	Format	Example in Prompt	Acc.
NNDC (Ours)	✓	✓	×	✓	**35.25**
w/o Clip.	✓	×	×	✓	14.8
w/o Norm.	×	✓	×	✓	0.00
w Format.	✓	✓	✓	✓	33.61
w/o Example.	✓	✓	×	×	17.21
NSDH	✓	✓	×	✓	21.31
w/o Clip.	✓	×	×	✓	22.31
w/o Norm.	×	✓	×	✓	0.00
NAC	✓	✓	×	✓	22.95
w/o Clip.	✓	×	×	✓	22.95
w/o Norm.	×	✓	×	✓	0.00
Binary	✓	✓	×	✓	15.57

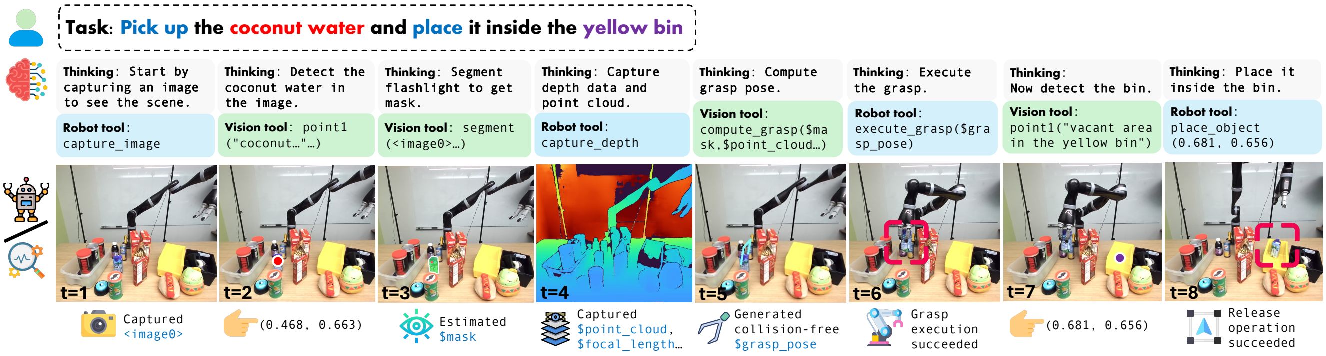
Dataset size and type for IRL. We conduct preliminary experiments on how dataset size and data-type composition affect IRL performance using the RoboSpatial dataset and the Roborefer pointing tool. RoboSpatial contains four data types: configuration, compatibility, grounding, and vacant-space localization. Configuration and compatibility are binary yes/no questions, grounding requires predicting 2D bounding boxes, and vacant-space localization involves predicting a free-space location. We vary the mixture of these four types and evaluate performance on RoboSpatial-Home, with results summarized in Table 10. Notably, for configuration and compatibility, we ensure a balanced distribution of yes/no answers, as discussed in Appendix D. Interestingly, although grounding data are not present in RoboSpatial-Home, including grounding during training improves performance on the other tasks. In contrast, increasing the overall dataset size beyond a moderate scale yields limited gains, suggesting that data diversity and label balance contribute more to IRL effectiveness than raw quantity alone.

Table 10. Evaluation on RoboSpatial-Home using models trained with Tool IRL under different data compositions drawn from the four RoboSpatial data types. *Config.* refers to configuration data, *Compat.* to compatibility data, *Ground.* to grounding (2D bounding box) data, and *Vacant* to vacant-space localization data. Each entry in the middle columns indicates the number of samples included for that data category. *Overall Acc.* reports the final accuracy on RoboSpatial-Home.

Variant	Config.	Compat.	Ground.	Vacant	Overall Acc.
All-v1	0.5k	0.5k	0.5k	0.5k	69.70
All-v2	1.0k	1.0k	1.0k	1.0k	69.70
All-v3	2.0k	2.0k	1.0k	1.0k	69.10
w/o Ground.	2.0k	2.0k	0.0k	2.0k	56.90

E.5. Additional Demonstrations

Visualizations of success cases. Apart from the cases where the grasp tool fails to find a collision-free pose or the pointing tool fails to localize points accurately for relative



Task: Pick up the coconut water and place it inside the yellow bin

Thinking: Start by capturing an image to see the scene.
Robot tool: `capture_image`

Thinking: Detect the coconut water in the image.
Vision tool: `point1("coconut…"…)`

Thinking: Segment flashlight to get mask.
Vision tool: `segment(<image0>…)`

Thinking: Capture depth data and point cloud.
Robot tool: `capture_depth`

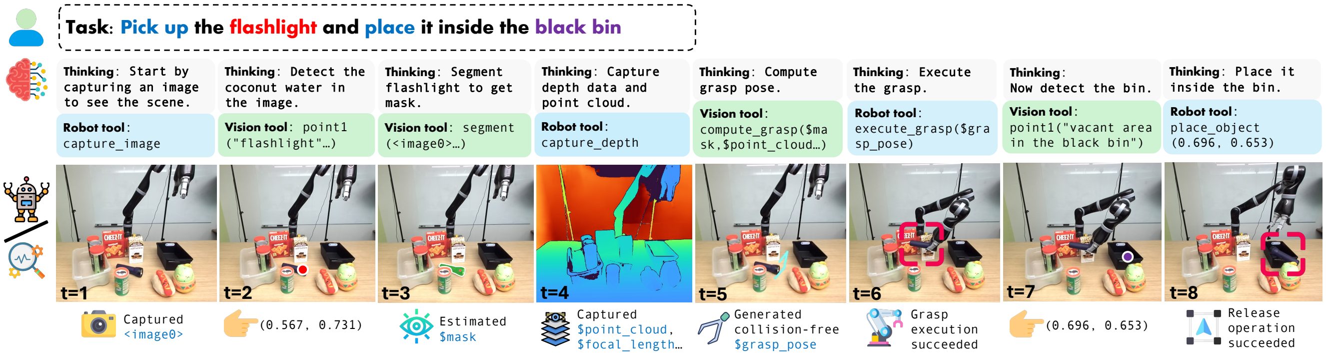
Thinking: Compute grasp pose.
Vision tool: `compute_grasp($mask,$point_cloud…)`

Thinking: Execute the grasp.
Robot tool: `execute_grasp($grasp_pose)`

Thinking: Now detect the bin.
Vision tool: `point1("vacant area in the yellow bin")`

Thinking: Place it inside the bin.
Robot tool: `place_object(0.681, 0.656)`

Figure 4. A hard real-world robot manipulation example with SpaceTools. The model successfully identifies the target object and completes the manipulation task in a cluttered and visually complex scene



Task: Pick up the flashlight and place it inside the black bin

Thinking: Start by capturing an image to see the scene.
Robot tool: `capture_image`

Thinking: Detect the coconut water in the image.
Vision tool: `point1("flashlight"…)`

Thinking: Segment flashlight to get mask.
Vision tool: `segment(<image0>…)`

Thinking: Capture depth data and point cloud.
Robot tool: `capture_depth`

Thinking: Compute grasp pose.
Vision tool: `compute_grasp($mask,$point_cloud…)`

Thinking: Execute the grasp.
Robot tool: `execute_grasp($grasp_pose)`

Thinking: Now detect the bin.
Vision tool: `point1("vacant area in the black bin")`

Thinking: Place it inside the bin.
Robot tool: `place_object(0.696, 0.653)`

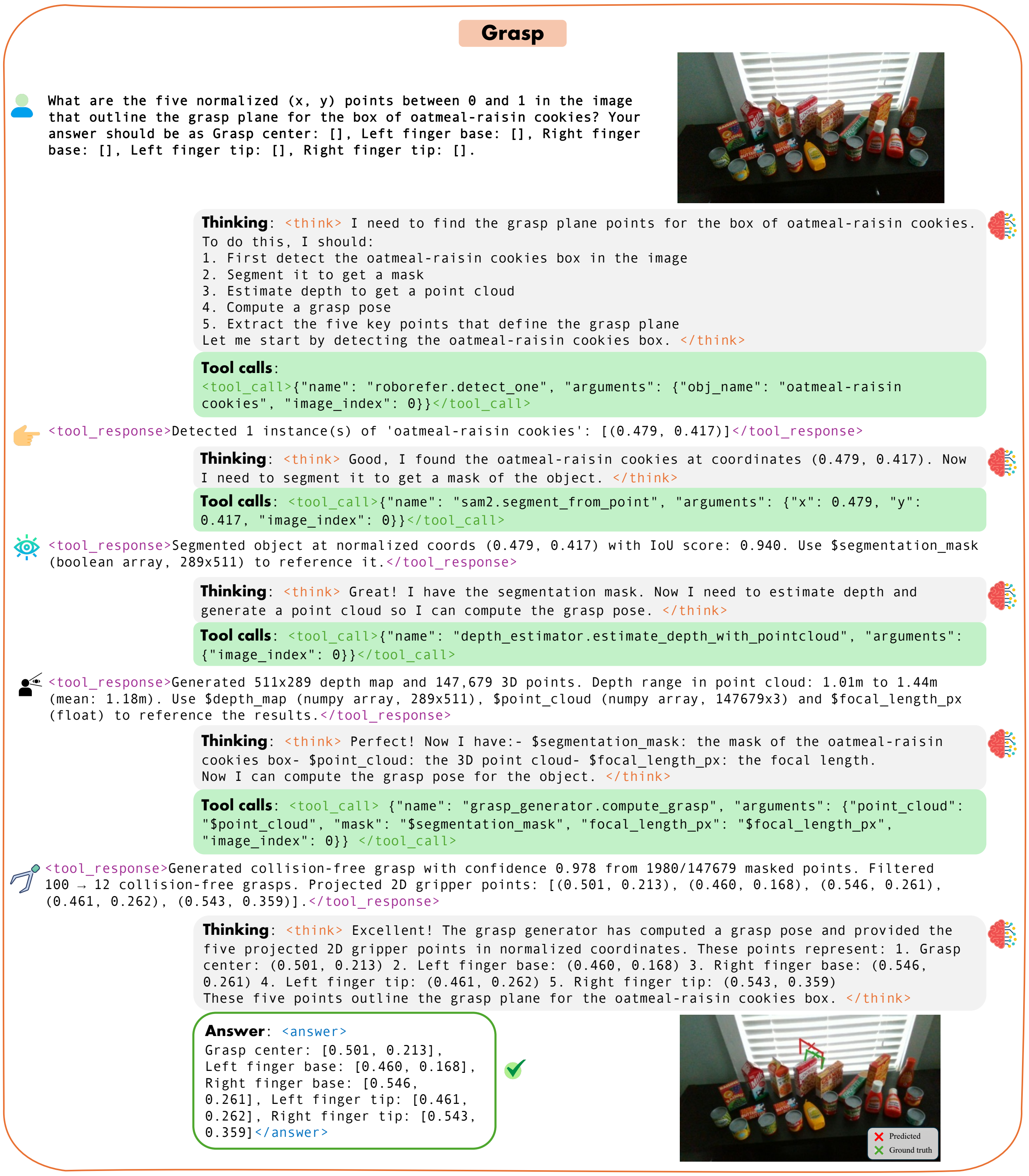
Figure 5. A failure case in real-world robot manipulation with SpaceTools. The model localizes a valid vacant area but selects a point too close to the boundary, resulting in a failed placement of the flashlight on the box boundary.

depth questions, we also present detailed examples of successful tool executions containing all special format-related tokens in Figure 6. In this example, the grasp tool successfully finds an accurate pose in a cluttered scene. Figure 7 provides examples for relative depth questions. One example shows the tool accurately predicts the required point locations, enabling the VLM to produce the correct answer. The other example shows multiple chairs are present in the image, and the VLM must identify the specific one highlighted by the provided bounding box. Moreover, as illustrated in Figure 4, SpaceTools can reliably identify target objects and execute the required manipulation steps even in cluttered, visually complex real-world environments.

Visualizations of failure cases. Although the model performs well across many scenarios, it is not universally reliable. We therefore examine representative failure cases to highlight remaining challenges and motivate future exploration. Grasp estimation currently yields the lowest accuracy, largely due to difficulties in detecting the target object in cluttered scenes and in predicting accurate yet collision-free grasp poses, as illustrated in Figure 8. Object detection in complex, visually crowded environments remains a major bottleneck for both grasp and pose estimation, leaving substantial room for improvement. These issues point not only to opportunities for improving individual tools but also to the potential of modular enhancements to spatial reasoning by strengthening specific tool components instead of retraining or finetuning the VLM.

Another failure example of real-world robot manipulation is shown in Figure 5. This failure case highlights a subtle but important challenge in real-world manipulation. Although the model correctly identifies a vacant region, it selects a point near the boundary, and thus, the robot finally performs the placement on the boundary. Such cases underscore the need for highly precise geometric reasoning and tighter coupling between 2D point predictions and physical feasibility on the robot. They also reveal a current limitation of the model's tool coordination and point-selection strategy, suggesting promising directions for future improvements, including integrating real or simulated robot feedback into the training process.

Quantitative failure analysis. We further report representative failure statistics on the grasp benchmark and real-world robot manipulation in Table 11. Specifically, we



Grasp

What are the five normalized (x, y) points between 0 and 1 in the image that outline the grasp plane for the box of oatmeal-raisin cookies? Your answer should be as Grasp center: [], Left finger base: [], Right finger base: [], Left finger tip: [], Right finger tip: [].

Thinking: <think> I need to find the grasp plane points for the box of oatmeal-raisin cookies.
To do this, I should:
1. First detect the oatmeal-raisin cookies box in the image
2. Segment it to get a mask
3. Estimate depth to get a point cloud
4. Compute a grasp pose
5. Extract the five key points that define the grasp plane
Let me start by detecting the oatmeal-raisin cookies box. </think>

Tool calls:
<tool_call>{"name": "roborefer.detect_one", "arguments": {"obj_name": "oatmeal-raisin cookies", "image_index": 0}}</tool_call>

<tool_response>Detected 1 instance(s) of 'oatmeal-raisin cookies': [(0.479, 0.417)]</tool_response>

Thinking: <think> Good, I found the oatmeal-raisin cookies at coordinates (0.479, 0.417). Now I need to segment it to get a mask of the object. </think>

Tool calls: <tool_call>{"name": "sam2.segment_from_point", "arguments": {"x": 0.479, "y": 0.417, "image_index": 0}}</tool_call>

<tool_response>Segmented object at normalized coords (0.479, 0.417) with IoU score: 0.940. Use $segmentation_mask (boolean array, 289x511) to reference it.</tool_response>

Thinking: <think> Great! I have the segmentation mask. Now I need to estimate depth and generate a point cloud so I can compute the grasp pose. </think>

Tool calls: <tool_call>{"name": "depth_estimator.estimate_depth_with_pointcloud", "arguments": {"image_index": 0}}</tool_call>

<tool_response>Generated 511x289 depth map and 147,679 3D points. Depth range in point cloud: 1.01m to 1.44m (mean: 1.18m). Use $depth_map (numpy array, 289x511), $point_cloud (numpy array, 147679x3) and $focal_length_px (float) to reference the results.</tool_response>

Thinking: <think> Perfect! Now I have:- $segmentation_mask: the mask of the oatmeal-raisin cookies box- $point_cloud: the 3D point cloud- $focal_length_px: the focal length.
Now I can compute the grasp pose for the object. </think>

Tool calls: <tool_call> {"name": "grasp_generator.compute_grasp", "arguments": {"point_cloud": "$point_cloud", "mask": "$segmentation_mask", "focal_length_px": "$focal_length_px", "image_index": 0}} </tool_call>

<tool_response>Generated collision-free grasp with confidence 0.978 from 1980/147679 masked points. Filtered 100 → 12 collision-free grasps. Projected 2D gripper points: [(0.501, 0.213), (0.460, 0.168), (0.546, 0.261), (0.461, 0.262), (0.543, 0.359)].</tool_response>

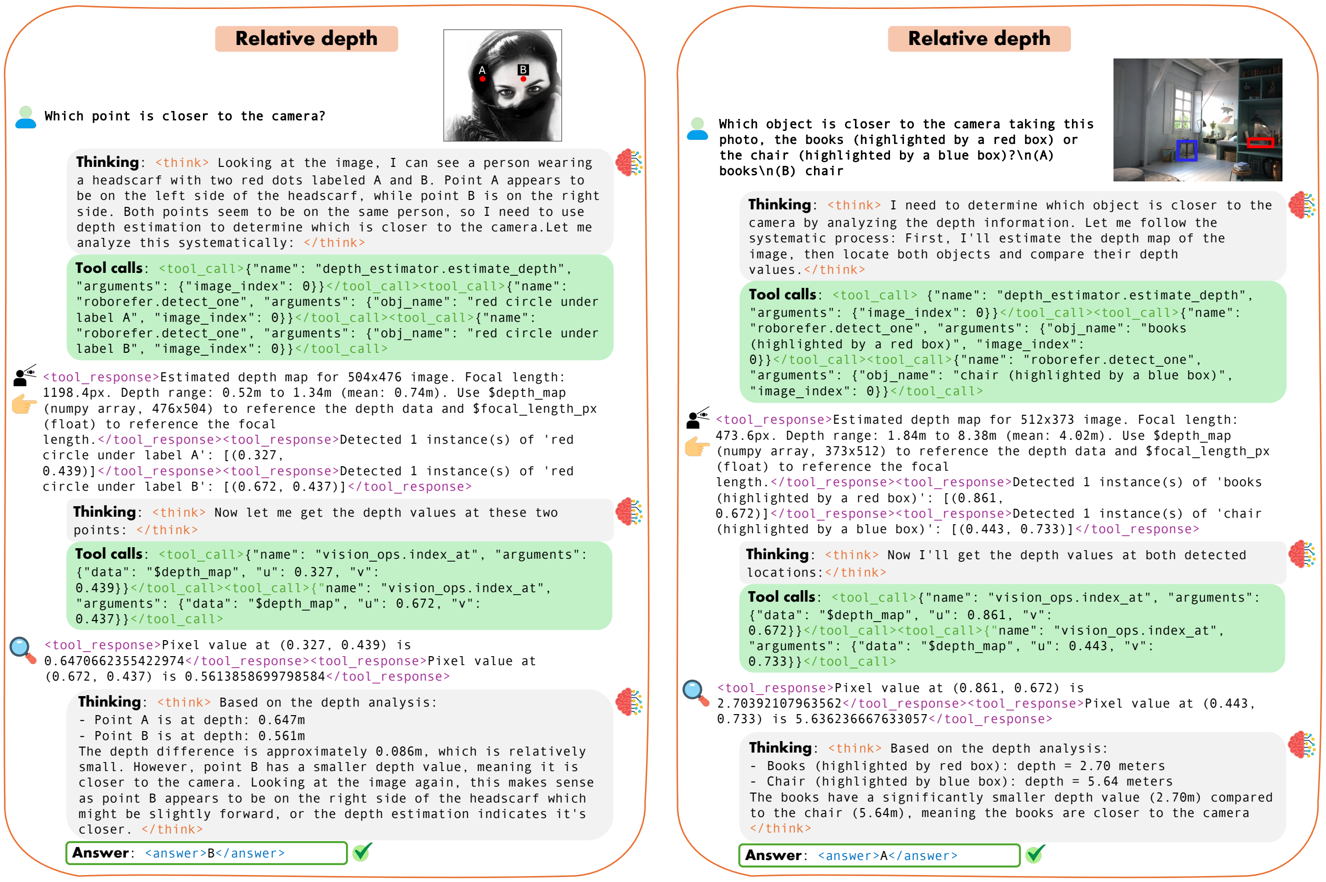
Thinking: <think> Excellent! The grasp generator has computed a grasp pose and provided the five projected 2D gripper points in normalized coordinates. These points represent: 1. Grasp center: (0.501, 0.213) 2. Left finger base: (0.460, 0.168) 3. Right finger base: (0.546, 0.261) 4. Left finger tip: (0.461, 0.262) 5. Right finger tip: (0.543, 0.359)
These five points outline the grasp plane for the oatmeal-raisin cookies box. </think>

Answer: <answer>
Grasp center: [0.501, 0.213], Left finger base: [0.460, 0.168], Right finger base: [0.546, 0.261], Left finger tip: [0.461, 0.262], Right finger tip: [0.543, 0.359]</answer>

✓

Figure 6. A detailed example of tool-augmented reasoning of a grasp estimation question.

analyze 30 failures out of 60 grasp benchmark trials and 4 failures out of 27 robot manipulation trials. Since execution traces are often intertwined—for example, a tool failure may trigger tool switching or fallback reasoning—



Relative depth

Which point is closer to the camera?

Thinking: <think> Looking at the image, I can see a person wearing a headscarf with two red dots labeled A and B. Point A appears to be on the left side of the headscarf, while point B is on the right side. Both points seem to be on the same person, so I need to use depth estimation to determine which is closer to the camera.Let me analyze this systematically: </think>

Tool calls:
```
<tool_call>{"name": "depth_estimator.estimate_depth", "arguments": {"image_index": 0}}</tool_call><tool_call>{"name": "roborefer.detect_one", "arguments": {"obj_name": "red circle under label A", "image_index": 0}}</tool_call><tool_call>{"name": "roborefer.detect_one", "arguments": {"obj_name": "red circle under label B", "image_index": 0}}</tool_call>
```

```
<tool_response>Estimated depth map for 504x476 image. Focal length: 1198.4px. Depth range: 0.52m to 1.34m (mean: 0.74m). Use $depth_map (numpy array, 476x504) to reference the depth data and $focal_length_px (float) to reference the focal length.</tool_response><tool_response>Detected 1 instance(s) of 'red circle under label A': [(0.327, 0.439)]</tool_response><tool_response>Detected 1 instance(s) of 'red circle under label B': [(0.672, 0.437)]</tool_response>
```

Thinking: <think> Now let me get the depth values at these two points: </think>

Tool calls:
```
<tool_call>{"name": "vision_ops.index_at", "arguments": {"data": "$depth_map", "u": 0.327, "v": 0.439}}</tool_call><tool_call>{"name": "vision_ops.index_at", "arguments": {"data": "$depth_map", "u": 0.672, "v": 0.437}}</tool_call>
```

```
<tool_response>Pixel value at (0.327, 0.439) is 0.6470662355422974</tool_response><tool_response>Pixel value at (0.672, 0.437) is 0.5613858699798584</tool_response>
```

Thinking: <think> Based on the depth analysis:
- Point A is at depth: 0.647m
- Point B is at depth: 0.561m
The depth difference is approximately 0.086m, which is relatively small. However, point B has a smaller depth value, meaning it is closer to the camera. Looking at the image again, this makes sense as point B appears to be on the right side of the headscarf which might be slightly forward, or the depth estimation indicates it's closer. </think>

Answer: <answer>B</answer> ✓

Relative depth

Which object is closer to the camera taking this photo, the books (highlighted by a red box) or the chair (highlighted by a blue box)?\n(A) books\n(B) chair

Thinking: <think> I need to determine which object is closer to the camera by analyzing the depth information. Let me follow the systematic process: First, I'll estimate the depth map of the image, then locate both objects and compare their depth values.</think>

Tool calls:
```
<tool_call> {"name": "depth_estimator.estimate_depth", "arguments": {"image_index": 0}}</tool_call><tool_call>{"name": "roborefer.detect_one", "arguments": {"obj_name": "books (highlighted by a red box)", "image_index": 0}}</tool_call><tool_call>{"name": "roborefer.detect_one", "arguments": {"obj_name": "chair (highlighted by a blue box)", "image_index": 0}}</tool_call>
```

```
<tool_response>Estimated depth map for 512x373 image. Focal length: 473.6px. Depth range: 1.84m to 8.38m (mean: 4.02m). Use $depth_map (numpy array, 373x512) to reference the depth data and $focal_length_px (float) to reference the focal length.</tool_response><tool_response>Detected 1 instance(s) of 'books (highlighted by a red box)': [(0.861, 0.672)]</tool_response><tool_response>Detected 1 instance(s) of 'chair (highlighted by a blue box)': [(0.443, 0.733)]</tool_response>
```

Thinking: <think> Now I'll get the depth values at both detected locations:</think>

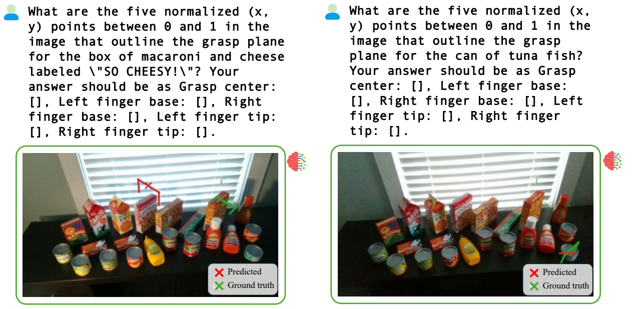
Tool calls:
```
<tool_call>{"name": "vision_ops.index_at", "arguments": {"data": "$depth_map", "u": 0.861, "v": 0.672}}</tool_call><tool_call>{"name": "vision_ops.index_at", "arguments": {"data": "$depth_map", "u": 0.443, "v": 0.733}}</tool_call>
```

```
<tool_response>Pixel value at (0.861, 0.672) is 2.70392107963562</tool_response><tool_response>Pixel value at (0.443, 0.733) is 5.636236667633057</tool_response>
```

Thinking: <think> Based on the depth analysis:
- Books (highlighted by red box): depth = 2.70 meters
- Chair (highlighted by blue box): depth = 5.64 meters
The books have a significantly smaller depth value (2.70m) compared to the chair (5.64m), meaning the books are closer to the camera </think>

Answer: <answer>A</answer> ✓

Figure 7. Detailed examples of tool-augmented reasoning of relative depth questions.



What are the five normalized (x, y) points between 0 and 1 in the image that outline the grasp plane for the box of macaroni and cheese labeled \"SO CHEESY!\"? Your answer should be as Grasp center: [], Left finger base: [], Right finger base: [], Left finger tip: [], Right finger tip: [].

What are the five normalized (x, y) points between 0 and 1 in the image that outline the grasp plane for the can of tuna fish? Your answer should be as Grasp center: [], Left finger base: [], Right finger base: [], Left finger tip: [], Right finger tip: [].

(a) Wrong object localization

(b) Inaccurate pose estimation

Figure 8. Failure cases for the grasp estimation task. Intermediate tool-augmented reasoning steps are omitted for clarity.

individual failures cannot always be uniquely attributed to a single source. We therefore categorize each failed case by the most direct cause that led to the final incorrect outcome.

Table 11. Representative failure breakdown by primary error source. We classify each failed case according to the most direct cause leading to the incorrect outcome.

Setting	Tool Error	Planning Error	Reasoning Error
Robot Manipulation	2	2	0
Grasp Benchmark	23	0	7

References

[1] Aleksei Bochkovskii, Amaël Delaunoy, Hugo Germain, Marcel Santos, Yichao Zhou, Stephan R. Richter, and Vladlen Koltun. Depth pro: Sharp monocular metric depth in less than a second. In *International Conference on Learning Representations*, 2025. 2

[2] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. 11

[3] Matt Deitke, Christopher Clark, Sangho Lee, Rohun Tripathi, Yue Yang, Jae Sung Park, Mohammadreza Salehi, Niklas Muennighoff, Kyle Lo, Luca Soldaini, Jiasen Lu, Taira Anderson, Erin Bransom, Kiana Ehsani, Huong Ngo, Yen-Sung Chen, Ajay Patel, Mark Yatskar, Chris Callison-Burch, Andrew Head, Rose Hendrix, Favyen Bastani, Eli VanderBilt, Nathan Lambert, Yvonne Chou, Arnavi Chheda, Jenna Sparks, Sam Skjonsberg, Michael Schmitz, Aaron Sarnat, Byron Bischoff, Pete Walsh, Chris Newell, Piper Wolters, Tanmay Gupta, Kuo-Hao Zeng, Jon Borchardt, Dirk Groen-

eveld, Crystal Nam, Sophie Lebrecht, Caitlin Wittlif, Carissa Schoenick, Oscar Michel, Ranjay Krishna, Luca Weihs, Noah A. Smith, Hannaneh Hajishirzi, Ross Girshick, Ali Farhadi, and Aniruddha Kembhavi. Molmo and pixmo: Open weights and open data for state-of-the-art vision-language models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025. 2

[4] Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, Eric Mintun, Junting Pan, Kalyan Vasudev Alwala, Nicolas Carion, Chao-Yuan Wu, Ross Girshick, Piotr Dollár, and Christoph Feichtenhofer. Sam 2: Segment anything in images and videos. *arXiv preprint arXiv:2408.00714*, 2024. 2

[5] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. 8

[6] Balakumar Sundaralingam, Siva Kumar Sastry Hari, Adam Fishman, Caelan Garrett, Karl Van Wyk, Valts Blukis, Alexander Millane, Helen Oleynikova, Ankur Handa, Fabio Ramos, Nathan Ratliff, and Dieter Fox. curobo: Parallelized collision-free minimum-jerk robot motion generation, 2023. 10

[7] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2022. Curran Associates Inc. 11

[8] Shaokun Zhang, Yi Dong, Jieyu Zhang, Jan Kautz, Bryan Catanzaro, Andrew Tao, Qingyun Wu, Zhiding Yu, and Guilin Liu. Nemotron-research-tool-n1: Exploring tool-using language models with reinforced reasoning, 2025. 10

[9] Enshen Zhou, Jingkun An, Cheng Chi, Yi Han, Shanyu Rong, Chi Zhang, Pengwei Wang, Zhongyuan Wang, Tiejun Huang, Lu Sheng, et al. Roborefer: Towards spatial referring with reasoning in vision-language models for robotics. *arXiv preprint arXiv:2506.04308*, 2025. 2