Visual Agentic AI for Spatial Reasoning with a Dynamic API

Supplementary Material

	Method	CLEVR	OMNI3D-BENCH
VLMs	GPT4o [1]	1.4	0.6
	Claude3.5-Sonnet [2]	0.2	0.6
	Llama3.2 [9]	0.5	1.6
	Gemini1.5-Pro [36]	0.3	1.8
	Gemini1.5-Flash [36]	0.3	1.1
	Molmo [8]	0.0	0.0
	SpaceMantis [6, 17]	0.0	0.0
Program Synthesis	ViperGPT [35]	1.1	0.3
	VisProg [12]	0.9	0.3
	VADAR (ours)	2.9	1.8

Table 5. **Standard deviation across experimental runs.** VADAR's variation is comparable to VLMs on Omni3D, but slightly higher than program synthesis methods on CLEVR, despite achieving significantly higher accuracy.

Signature (for 10 Qs)	Implementation	Program (per Q)	Execution (per Q)
$20.5_{\pm 3.6}$	$37.2_{\pm 14.4}$	$6.5_{\pm 1.8}$	$35.7_{\pm 11.8}$

Table 6. **Runtime for each Agent in seconds.**

The Appendix includes the prompts used for all agents, additional qualitative examples of VADAR on CLEVR, OMNI3D-BENCH, and GQA, and a supplemental qualitative analysis with standard deviations to compare the robustness of approaches.

A. Prompts

Predefined Module Signatures. Fig. 9 and Fig. 10 show the docstrings of the predefined modules for CLEVR and OMNI3D-BENCH respectively, which are used to initialize the dynamic API. We note that the two prompts are virtually identical, with the exception of the `get_2D_object_size` method, which we omit from our experiments on CLEVR as the dataset defines size as either `small` or `large`. In Fig. 11, we provide the Python implementation for all of the predefined modules.

Signature Agent Prompt. Fig. 12 contains the prompt used for the Signature Agent for both CLEVR and OMNI3D-BENCH. We prompt the LLM to only generate signatures for methods when necessary, as we found this avoids redundant methods with minor changes to previously defined methods. We impose that the name of new methods start with an underscore, to prevent the common failure case of methods sharing names with variables previously defined.

Implementation Agent Prompt. Fig. 13 and Fig. 14 contain the prompts used for the Implementation agent on CLEVR and OMNI3D-BENCH respectively. The prompts contain *Weak ICL* examples, illustrating how to implement a model signature and use the pre-defined modules correctly for simpler queries. This is in contrast to *Strong ICL* examples in VisProg and ViperGPT, which provide complete program examples for full queries using a predefined API. In our framework, where agents dynamically generate the API, *Strong ICL* is not feasible.

Additionally, the prompts feature *Pseudo ICL* in the form of natural language instructions and tips. Similarly to the predefined modules, the prompts differ between CLEVR and OMNI3D-BENCH as the latter considers metric sizes and not a binary `small` or `large` as in CLEVR. Consequently, we found it necessary to include natural language definitions and instructions for reasoning about 2D and 3D dimensions in the Implementation prompt on OMNI3D-BENCH.

Program Agent Prompt. In Fig. 15 and Fig. 16 we show the prompts for the Program Agent on CLEVR and OMNI3D-BENCH respectively. In the prompt for CLEVR, we include a list of all available attributes. In both prompts, we include *Pseudo ICL* in the form of natural language examples and instructions. For the OMNI3D-BENCH prompt, we additionally include tips and definitions for handling 2D and 3D dimensions.

B. Additional Quantitative Analysis

Experimental Variability. Tab. 1 in the main paper reports the mean performance of all methods across 3 runs. Tab. 5 reports the standard deviation on CLEVR and OMNI3D-BENCH across the same 3 runs. VADAR's variation is comparable to the VLMs on OMNI3D-BENCH, but slightly higher than program synthesis methods on both benchmarks. However, VADAR significantly outperforms ViperGPT and VisProg, even when accounting for this variation.

Runtime. Tab. 6 reports runtime in seconds for our Agents on an A100 GPU. Notably, when running our method on 1000+ questions, the Signature and Implementation Agents *only run once*, therefore their runtime becomes negligible to total inference runtime.

C. More information on OMNI3D-BENCH

On images sourced from Omni3D [5] we collect a set of challenging questions with the help of human annotators. We omit using templates for questions, as done by others [6, 38, 44], to avoid template overfitting, and instead instruct annotators to directly ask questions in free-form natural language, focusing on the scene, object layout and

	VSI-Bench-img
Gemini1.5-Pro	49.5
VADAR	**50.1**

Table 7. **Results on VSI-Bench [44].** VADAR outperforms Gemini1.5-Pro on a image-based subset of 75 queries from VSI-Bench that sources the frame that contains all the information necessary to respond correctly. Notably, VADAR achieves a 50.1% accuracy on this subset, compared to 40.4% on OMNI3D-BENCH, highlighting the challenging nature of our proposed benchmark.

object sizes. We discard questions that are simplistic, *e.g.* "Is there a sofa in the image?" or "Is the sofa behind the table?", and only keep queries which involve complex inference steps in 2D and 3D. OMNI3D-BENCH queries roughly target the following areas of reasoning: relative size and dimensions with hypotheticals, spatial relationships and depth reasoning, relative proportions and alignments, and interaction with other objects. Queries from OMNI3D-BENCH can be browsed in https://glab-caltech.github.io/vadar/omni3d-bench.html.

We compute answers for questions using the 3D annotations provided in Omni3D [5]. Since the questions are not templated and thus don't follow rule-based instructions, we collect answers manually by sourcing the 3D annotations provided by the dataset for each image. This results in 500 *unique* and challenging image-question-answer tuples that test diverse aspects of 3D spatial reasoning. The diversity and complexity of OMNI3D-BENCH is showcased by the examples in Fig. 1, Fig. 4 and Fig. 7.

OMNI3D-BENCH complements CLEVR when assessing 3D spatial understanding. While CLEVR uses templated questions, enabling the creation of a large volume of image-question-answer pairs, OMNI3D-BENCH focuses on diverse and complex reasoning tasks in free-form language. Together, CLEVR and OMNI3D-BENCH provide a comprehensive test for models' 3D spatial reasoning capabilities. This is evidenced by the relatively low performance of modern state-of-the-art AI models on these benchmarks, achieving only 20-40% accuracy.

D. Comparison to VSI-Bench

Concurrent to our work is VSI-Bench [44], a video understanding benchmark that focuses on spatial reasoning. VSI-Bench targets 3D reasoning, but it differs from OMNI3D-BENCH in three critical ways: First, it focuses on video understanding and retrieving the appropriate frame to answer a given query. Second, while queries in VSI-Bench target 3D object attributes, they query absolute measurements, such as *"What is the height of the chair?"*. Monolithic VLMs when prompted with such questions resort to object priors. For example, GPT4o says: *"A chair tends to be 30-40 inches tall"*. In contrast, OMNI3D-BENCH introduces hypotheticals that require reasoning over scene attributes, evaluating true 3D spatial reasoning, *e.g.*, *"If the table is 2 meters wide, how tall is the chair?"*. Third, VSI-Bench queries are templated, which can lead to biased conclusions due to template overfitting.

We compare VADAR on VSI-Bench. To decouple frame retrieval from image-based reasoning, we create a variant of the benchmark by sourcing a subset of 75 queries with the associated frame that contains the information necessary to address the query. We call this subset VSI-Bench-img. Tab. 7 reports VADAR's performance and compares to Gemini1.5-Pro, which authors report to be the best VLM on the set. From Tab. 7 we observe that VADAR performs on par with the industry-leading Gemini1.5-pro. Importantly, VADAR's performance on VSI-Bench-img is 10% higher than on OMNI3D-BENCH (40.4 vs 50.1) which highlights the more challenging nature of our benchmark.

E. Qualitative Examples on CLEVR

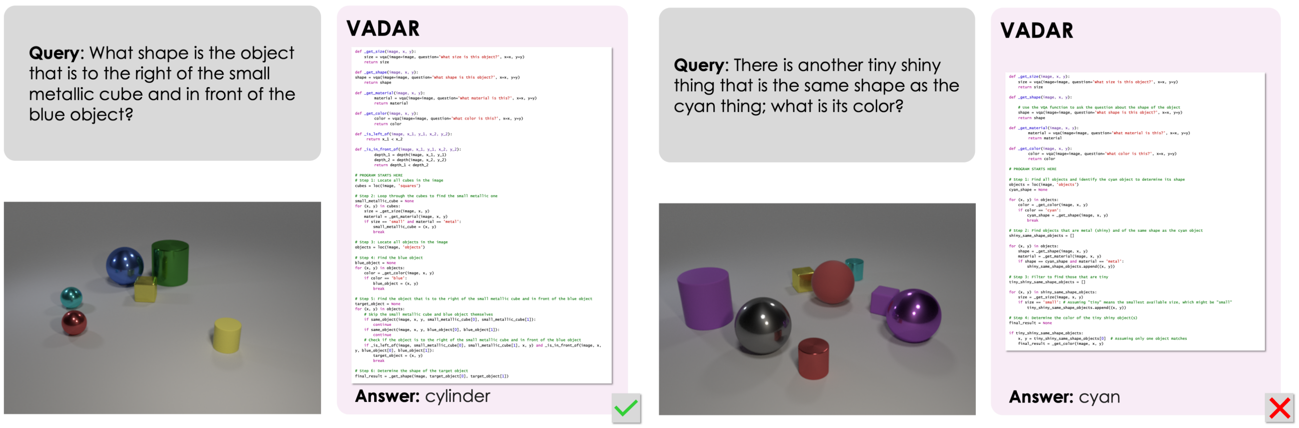
Fig. 6 shows additional qualitative examples on CLEVR. The correct example showcases the use of API methods for repeated tasks and accurately determining spatial relations. The incorrect example highlights a failure to use same object to exclude the original reference object when the questions asks for "another" object.

F. Qualitative Examples on OMNI3D-BENCH

Fig. 7 shows additional qualitative examples on OMNI3D-BENCH. Our method is able to correctly estimate 3D distances by scaling depth based on the reference scale given in the question. An instance where such scaling is done incorrectly is shown in the last example.

G. Qualitative Examples on GQA

Fig. 8 shows qualitative examples on GQA [16]. Our method is able to identify and locate key objects necessary to answer questions. It is extremely explicit, locating the nearest person in the top right example using pixel distance from the tree. Some GQA questions have ambiguous answers, where the shape of the pot is generically "round" and the frame of reference for spatial relations is not entirely clear (*i.e.*, which man in the last example?).



Query: What shape is the object that is to the right of the small metallic cube and in front of the blue object?

VADAR

```python
def _get_size(image, x, y):
    size = vqa(image=image, question='What size is this object?', x=x, y=y)
    return size

def _get_shape(image, x, y):
shape = vqa(image=image, question='What shape is this object?', x=x, y=y)
    return shape

def _get_material(image, x, y):
        material = vqa(image=image, question='What material is this?', x=x, y=y)
        return material

def _get_color(image, x, y):
        color = vqa(image=image, question='What color is this?', x=x, y=y)
        return color

def _is_left_of(image, x_1, y_1, x_2, y_2):
     return x_1 < x_2

def _is_in_front_of(image, x_1, y_1, x_2, y_2):
        depth_1 = depth(image, x_1, y_1)
        depth_2 = depth(image, x_2, y_2)
        return depth_1 < depth_2

# PROGRAM STARTS HERE
# Step 1: Locate all cubes in the image
cubes = loc(image, 'squares')

# Step 2: Loop through the cubes to find the small metallic one
small_metallic_cube = None
for (x, y) in cubes:
    size = _get_size(image, x, y)
    material = _get_material(image, x, y)
    if size == 'small' and material == 'metal':
        small_metallic_cube = (x, y)
        break

# Step 3: Locate all objects in the image
objects = loc(image, 'objects')

# Step 4: Find the blue object
blue_object = None
for (x, y) in objects:
    color = _get_color(image, x, y)
    if color == 'blue':
        blue_object = (x, y)
        break

# Step 5: Find the object that is to the right of the small metallic cube and in front of the blue object
target_object = None
for (x, y) in objects:
    # Skip the small metallic cube and blue object themselves
    if same_object(image, x, y, small_metallic_cube[0], small_metallic_cube[1]):
        continue
    if same_object(image, x, y, blue_object[0], blue_object[1]):
        continue
    # Check if the object is to the right of the small metallic cube and in front of the blue object
    if _is_left_of(image, small_metallic_cube[0], small_metallic_cube[1], x, y) and _is_in_front_of(image, x,
y, blue_object[0], blue_object[1]):
        target_object = (x, y)
        break

# Step 6: Determine the shape of the target object
final_result = _get_shape(image, target_object[0], target_object[1])
```

Answer: cylinder ✓

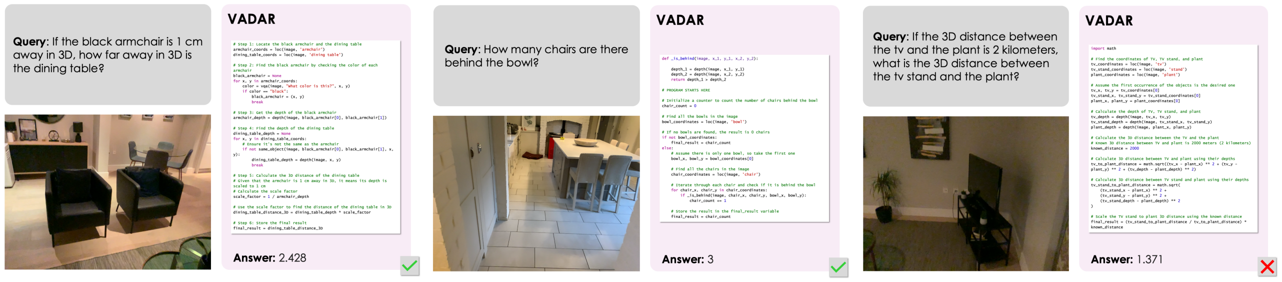
Query: There is another tiny shiny thing that is the same shape as the cyan thing; what is its color?

VADAR

```python
def _get_size(image, x, y):
    size = vqa(image=image, question='What size is this object?', x=x, y=y)
    return size

def _get_shape(image, x, y):

    # Use the VQA function to ask the question about the shape of the object
    shape = vqa(image=image, question='What shape is this object?', x=x, y=y)
    return shape

def _get_material(image, x, y):
        material = vqa(image=image, question='What material is this?', x=x, y=y)
        return material

def _get_color(image, x, y):
        color = vqa(image=image, question='What color is this?', x=x, y=y)
        return color

# PROGRAM STARTS HERE

# Step 1: Find all objects and identify the cyan object to determine its shape
objects = loc(image, 'objects')
cyan_shape = None

for (x, y) in objects:
    color = _get_color(image, x, y)
    if color == 'cyan':
        cyan_shape = _get_shape(image, x, y)
        break

# Step 2: Find objects that are metal (shiny) and of the same shape as the cyan object
shiny_same_shape_objects = []

for (x, y) in objects:
    shape = _get_shape(image, x, y)
    material = _get_material(image, x, y)
    if shape == cyan_shape and material == 'metal':
        shiny_same_shape_objects.append((x, y))

# Step 3: Filter to find those that are tiny
tiny_shiny_same_shape_objects = []

for (x, y) in shiny_same_shape_objects:
    size = _get_size(image, x, y)
    if size == 'small': # Assuming "tiny" means the smallest available size, which might be "small"
        tiny_shiny_same_shape_objects.append((x, y))

# Step 4: Determine the color of the tiny shiny object(s)
final_result = None

if tiny_shiny_same_shape_objects:
    x, y = tiny_shiny_same_shape_objects[0]  # Assuming only one object matches
    final_result = _get_color(image, x, y)
```

Answer: cyan ✗

Figure 6. VADAR program outputs on CLEVR.



Query: If the black armchair is 1 cm away in 3D, how far away in 3D is the dining table?

VADAR

```python
# Step 1: Locate the black armchair and the dining table
armchair_coords = loc(image, 'armchair')
dining_table_coords = loc(image, 'dining table')

# Step 2: Find the black armchair by checking the color of each
armchair
black_armchair = None
for x, y in armchair_coords:
    color = vqa(image, "What color is this?", x, y)
    if color == "black":
        black_armchair = (x, y)
        break

# Step 3: Get the depth of the black armchair
armchair_depth = depth(image, black_armchair[0], black_armchair[1])

# Step 4: Find the depth of the dining table
dining_table_depth = None
for x, y in dining_table_coords:
    # Ensure it's not the same as the armchair
    if not same_object(image, black_armchair[0], black_armchair[1], x,
y):
        dining_table_depth = depth(image, x, y)
        break

# Step 5: Calculate the 3D distance of the dining table
# Given that the armchair is 1 cm away in 3D, it means its depth is
scaled to 1 cm
# Calculate the scale factor
scale_factor = 1 / armchair_depth

# Use the scale factor to find the distance of the dining table in 3D
dining_table_distance_3D = dining_table_depth * scale_factor

# Step 6: Store the final result
final_result = dining_table_distance_3D
```

Answer: 2.428 ✓

Query: How many chairs are there behind the bowl?

VADAR

```python
def _is_behind(image, x_1, y_1, x_2, y_2):

    depth_1 = depth(image, x_1, y_1)
    depth_2 = depth(image, x_2, y_2)
    return depth_1 > depth_2

# PROGRAM STARTS HERE

# Initialize a counter to count the number of chairs behind the bowl
chair_count = 0

# Find all the bowls in the image
bowl_coordinates = loc(image, 'bowl')

# If no bowls are found, the result is 0 chairs
if not bowl_coordinates:
    final_result = chair_count
else:
    # Assume there is only one bowl, so take the first one
    bowl_x, bowl_y = bowl_coordinates[0]

    # Find all the chairs in the image
    chair_coordinates = loc(image, 'chair')

    # Iterate through each chair and check if it is behind the bowl
    for chair_x, chair_y in chair_coordinates:
        if _is_behind(image, chair_x, chair_y, bowl_x, bowl_y):
            chair_count += 1

    # Store the result in the final_result variable
    final_result = chair_count
```

Answer: 3 ✓

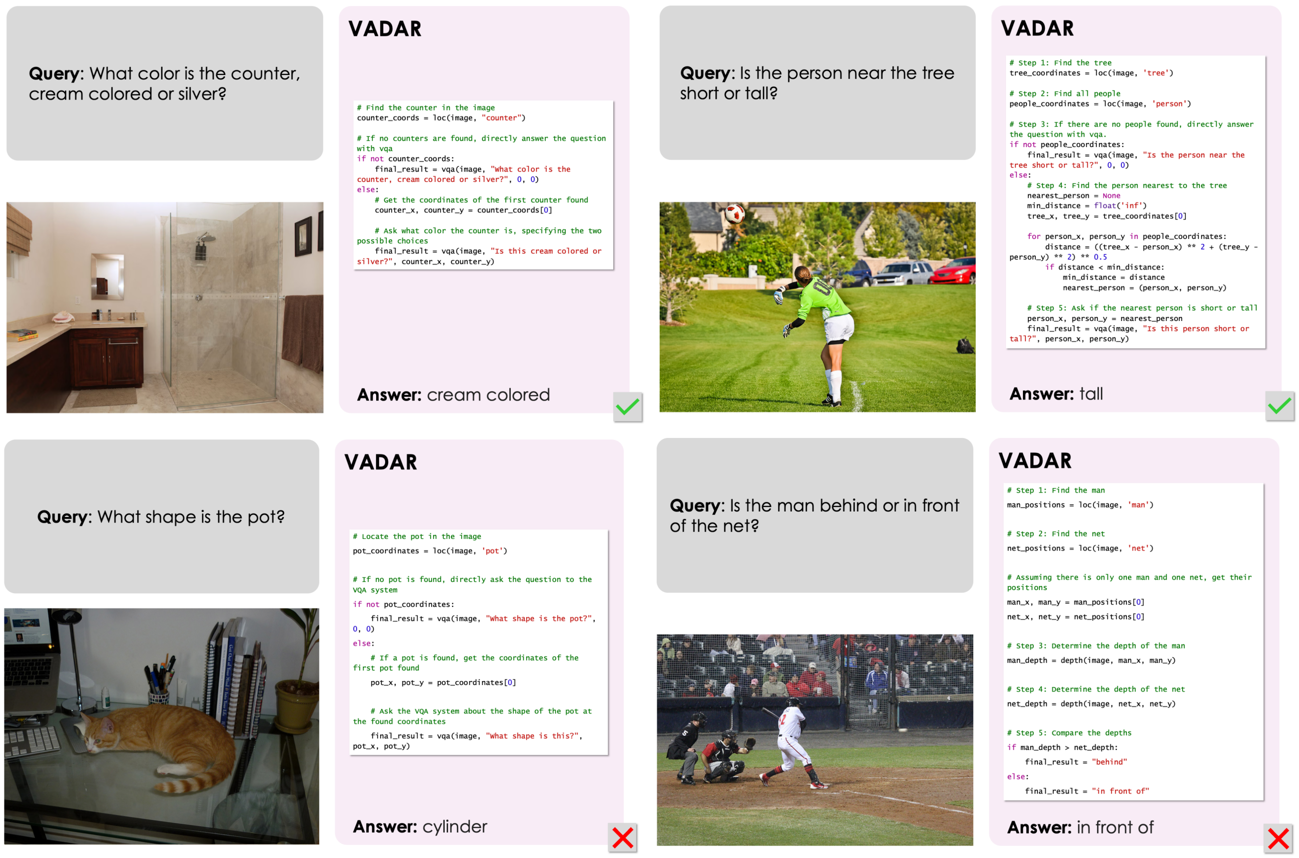
Query: If the 3D distance between the tv and the plant is 2 kilometers, what is the 3D distance between the tv stand and the plant?

VADAR

```python
import math

# Find the coordinates of TV, TV stand, and plant
tv_coordinates = loc(image, 'tv')
tv_stand_coordinates = loc(image, 'stand')
plant_coordinates = loc(image, 'plant')

# Assume the first occurrence of the objects is the desired one
tv_x, tv_y = tv_coordinates[0]
tv_stand_x, tv_stand_y = tv_stand_coordinates[0]
plant_x, plant_y = plant_coordinates[0]

# Calculate the depth of TV, TV stand, and plant
tv_depth = depth(image, tv_x, tv_y)
tv_stand_depth = depth(image, tv_stand_x, tv_stand_y)
plant_depth = depth(image, plant_x, plant_y)

# Calculate the 3D distance between the TV and the plant
# Known 3D distance between TV and plant is 2000 meters (2 kilometers)
known_distance = 2000

# Calculate 3D distance between TV and plant using their depths
tv_to_plant_distance = math.sqrt((tv_x - plant_x) ** 2 + (tv_y -
plant_y) ** 2 + (tv_depth - plant_depth) ** 2)

# Calculate 3D distance between TV stand and plant using their depths
tv_stand_to_plant_distance = math.sqrt(
    (tv_stand_x - plant_x) ** 2 +
    (tv_stand_y - plant_y) ** 2 +
    (tv_stand_depth - plant_depth) ** 2
)

# Scale the TV stand to plant 3D distance using the known distance
final_result = (tv_stand_to_plant_distance / tv_to_plant_distance) *
known_distance
```

Answer: 1.371 ✗

Figure 7. VADAR program outputs on OMNI3D-BENCH.



Query: What color is the counter, cream colored or silver?

VADAR

```python
# Find the counter in the image
counter_coords = loc(image, "counter")

# If no counters are found, directly answer the question
with vqa
if not counter_coords:
    final_result = vqa(image, "What color is the
counter, cream colored or silver?", 0, 0)
else:
    # Get the coordinates of the first counter found
    counter_x, counter_y = counter_coords[0]

    # Ask what color the counter is, specifying the two
possible choices
    final_result = vqa(image, "Is this cream colored or
silver?", counter_x, counter_y)
```

Answer: cream colored ✓

Query: Is the person near the tree short or tall?

VADAR

```python
# Step 1: Find the tree
tree_coordinates = loc(image, 'tree')

# Step 2: Find all people
people_coordinates = loc(image, 'person')

# Step 3: If there are no people found, directly answer
the question with vqa.
if not people_coordinates:
    final_result = vqa(image, "Is the person near the
tree short or tall?", 0, 0)
else:
    # Step 4: Find the person nearest to the tree
    nearest_person = None
    min_distance = float('inf')
    tree_x, tree_y = tree_coordinates[0]

    for person_x, person_y in people_coordinates:
        distance = ((tree_x - person_x) ** 2 + (tree_y -
person_y) ** 2) ** 0.5
        if distance < min_distance:
            min_distance = distance
            nearest_person = (person_x, person_y)

    # Step 5: Ask if the nearest person is short or tall
    person_x, person_y = nearest_person
    final_result = vqa(image, "Is this person short or
tall?", person_x, person_y)
```

Answer: tall ✓

Query: What shape is the pot?

VADAR

```python
# Locate the pot in the image
pot_coordinates = loc(image, 'pot')

# If no pot is found, directly ask the question to the
VQA system
if not pot_coordinates:
    final_result = vqa(image, "What shape is the pot?",
0, 0)
else:
    # If a pot is found, get the coordinates of the
first pot found
    pot_x, pot_y = pot_coordinates[0]

    # Ask the VQA system about the shape of the pot at
the found coordinates
    final_result = vqa(image, "What shape is this?",
pot_x, pot_y)
```

Answer: cylinder ✗

Query: Is the man behind or in front of the net?

VADAR

```python
# Step 1: Find the man
man_positions = loc(image, 'man')

# Step 2: Find the net
net_positions = loc(image, 'net')

# Assuming there is only one man and one net, get their
positions
man_x, man_y = man_positions[0]
net_x, net_y = net_positions[0]

# Step 3: Determine the depth of the man
man_depth = depth(image, man_x, man_y)

# Step 4: Determine the depth of the net
net_depth = depth(image, net_x, net_y)

# Step 5: Compare the depths
if man_depth > net_depth:
    final_result = "behind"
else:
    final_result = "in front of"
```

Answer: in front of ✗

Figure 8. VADAR program outputs on GQA [16].

```
\"\"\"
Locates objects in an image. Object prompts should be 1 WORD MAX.

Args:
   image (image): Image to search.
   object_prompt (string): Description of object to locate. Examples: "spheres", "objects".
Returns:
   list: A list of x,y coordinates for all of the objects located in pixel space.
\"\"\"
def loc(image, object_prompt):

\"\"\"
Answers a question about the attributes of an object specified by an x,y coordinate.
Should not be used for other kinds of questions.

Args:
   image (image): Image of the scene.
   question (string): Question about the objects attribute to answer. Examples: "What color is this?", "What material is this?"
   x (int): X coordinate of the object in pixel space.
   y (int): Y coordinate of the object in pixel space.


Returns:
   string: Answer to the question about the object in the image.
\"\"\"
def vqa(image, question, x, y):

\"\"\"
Returns the depth of an object specified by an x,y coordinate.

Args:
   image (image): Image of the scene.
   x (int): X coordinate of the object in pixel space.
   y (int): Y coordinate of the object in pixel space.

Returns:
   float: The depth of the object specified by the coordinates.
\"\"\"
def depth(image, x, y):

\"\"\"
Checks if two pairs of coordinates correspond to the same object.

Args:
   image (image): Image of the scene.
   x_1 (int): X coordinate of object 1 in pixel space.
   y_1 (int): Y coordinate of object 1 in pixel space.
   x_2 (int): X coordinate of object 2 in pixel space.
   y_2 (int): Y coordinate of object 2 in pixel space.

Returns:
   bool: True if object 1 is the same object as object 2, False otherwise.
\"\"\"
def same_object(image, x_1, y_1, x_2, y_2):
```

Figure 9. **Pre-defined Modules for CLEVR**. These modules are used to initialize the dynamic API. As CLEVR defines size to be either `large` or `small`, we omit the `get_2D_object_size` method.

```
\"\"\"
Locates objects in an image. Object prompts should be 1 WORD MAX.

Args:
   image (image): Image to search.
   object_prompt (string): Description of object to locate.
Returns:
   list: A list of x,y coordinates for all of the objects located in pixel space.
\"\"\"
def loc(image, object_prompt):

\"\"\"
Answers a question about the attributes of an object specified by an x,y coordinate.
Should not be used for other kinds of questions.

Args:
   image (image): Image of the scene.
   question (string): Question about the objects attribute to answer. Examples: "What color is this?", "What material is this?"
   x (int): X coordinate of the object in pixel space.
   y (int): Y coordinate of the object in pixel space.


Returns:
   string: Answer to the question about the object in the image.
\"\"\"
def vqa(image, question, x, y):

\"\"\"
Returns the depth of an object specified by an x,y coordinate.

Args:
   image (image): Image of the scene.
   x (int): X coordinate of the object in pixel space.
   y (int): Y coordinate of the object in pixel space.

Returns:
   float: The depth of the object specified by the coordinates.
\"\"\"
def depth(image, x, y):

\"\"\"
Checks if two pairs of coordinates correspond to the same object.

Args:
   image (image): Image of the scene.
   x_1 (int): X coordinate of object 1 in pixel space.
   y_1 (int): Y coordinate of object 1 in pixel space.
   x_2 (int): X coordinate of object 2 in pixel space.
   y_2 (int): Y coordinate of object 2 in pixel space.

Returns:
   bool: True if object 1 is the same object as object 2, False otherwise.
\"\"\"
def same_object(image, x_1, y_1, x_2, y_2):

\"\"\"
Returns the width and height of the object in 2D pixel space.

Args:
   image (image): Image of the scene.
   x (int): X coordinate of the object in pixel space.
   y (int): Y coordinate of the object in pixel space.

Returns:
   tuple: (width, height) of the object in 2D pixel space.
\"\"\"
def get_2D_object_size(image, x, y):
```

Figure 10. **Pre-defined Modules for OMNI3D-BENCH**. These modules are used to initialize the dynamic API.

```
def loc(self, image, object_prompt):
   pts = molmo(image, "point to the " + object_prompt)
   if len(pts) == 0:
      # No points found
      return []
   return pts

def vqa(image, question, x, y):
   mask = sam_2([x, y], "foreground") # get sam2 mask at x,y
   bbox = bbox_from_mask(mask) # bbox around sam2 mask
   boxed_image = overlay_box_on_image(image, bbox) # original image with bbox overlaid
   result = gpt4o(boxed_image, question)
   return result

def depth(image, x, y):
   depth_pred = unidepth(image)["depth"] # Predict depth map over image
   depth_x_y = depth_pred[y, x]
   return depth_x_y

def same_object(image, x_1, y_1, x_2, y_2):
   mask_1 = sam_2([x_1, y_1], "foreground") # get sam2 mask for point 1
   mask_2 = sam_2([x_2, y_2], "foreground") # get sam2 mask for point 2
   obj_1_bbox = bbox_from_mask(mask_1) # bbox around sam2 mask
   obj_2_bbox = bbox_from_mask(mask_2) # bbox around sam2 mask
   return iou(obj_1_bbox, obj_2_bbox) > 0.92

def get_2D_object_size(image, x, y):
   mask = sam_2([x, y], "foreground") # get sam2 mask at x,y
   bbox = bbox_from_mask(mask) # bbox around sam2 mask
   width = abs(box[0] - box[2])
   height = abs(box[1] - box[3])
   return width, height
```

Figure 11. **Python Implementation of Predefined Modules.** VADAR uses Molmo [8] for object detection, SAM2 [22] for segmentation, GPT4o [1] for VQA, and UniDepth [31] for depth estimation.

```
Propose only new method signatures to add to the existing API.

Available Primitives: image, int, string, list, tuple

Current API:
{current_api_signatures}

Next, I will ask you a series of questions that reference an image and are solvable with a python program that uses
the API I have provided so far. Please propose new method signatures with associated docstrings to add to the API that
 would help modularize the programs that answer the questions.

For each proposed method, output the docstring inside <docstring></docstring> immediately followed by the method
signature for the docstring inside <signature></signature>. Do not propose methods that are already in the API.

Please ensure that you ONLY add new methods when necessary. Do not add new methods if you can solve the problem with
combinations of the previous methods!

Added methods should be simple, building minorly on the methods that already exist.

Importantly, new methods MUST start with an underscore. As an example, you may define a "_get_material" method. Please
 ensure you ALWAYS start the name with an underscore.

Again, output the docstring inside <docstring></docstring> immediately followed by the method signature for the
docstring inside <signature></signature>.

{questions}
```

Figure 12. **Signature Agent Prompt** used for both CLEVR and OMNI3D-BENCH.

```
Implement a method given a docstring and method signature, using the API specification as necessary.
Current API:
{pre_defined_signatures}
{generated_signatures}

Here are some examples of how to implement a method given its docstring and signature:
<docstring>
\"\"\"
Locates objects that are on the left of the reference object.
Args:
   image (IMAGE): Image to search.
   ref_x (int): X coordinate of reference object in pixel space.
   ref_y (int): Y coordinate of reference object in pixel space.
Returns:
   points (list): list of [x, y] coordinates for objects in pixel space matching description to the left.
\"\"\"
</docstring>
<signature>def objects_left(image, ref_x, ref_y):</signature>
<implementation>
objects_left = []
all_objects = loc(image, object_prompt=’objects’)
for object_point in all_objects:
   x, y = object_point
   if same_object(image, ref_x, ref_y, x, y):
      continue
   if x < ref_x:
      objects_left.append(object_point)
return objects_left
</implementation>
<docstring>
\"\"\"
Gets the material of the given object.
Args:
   image (IMAGE): Image that the object is contained in.
   ref_x (int): X coordinate of reference object in pixel space.
   ref_y (int): Y coordinate of reference object in pixel space.
Returns:
   str: Material of the object.
\"\"\"
</docstring>
<signature>def object_material(image, ref_x, ref_y):</signature>
<implementation>
material = vqa(image=image, question=’What material is this object?’, x=ref_x, y=ref_y)
return material
</implementation>
<docstring>
\"\"\"
Checks if an object 1 is in front of object 2.
Args:
   image (IMAGE): Image that the object is contained in.
   x_1 (int): X coordinate of object 1 in pixel space.
   y_1 (int): Y coordinate of object 1 in pixel space.
   x_2 (int): X coordinate of object 2 in pixel space.
   y_2 (int): Y coordinate of object 2 in pixel space.
Returns:
   bool: True if object 1 is in front of object 2, False otherwise
\"\"\"
</docstring>
<signature>def in_front_of(image, x_1, y_1, x_2, y_2):</signature>
<implementation>
depth_1 = depth(image, x_1, y_1)
depth_2 = depth(image, x_2, y_2)
return depth_1 < depth_2
</implementation>
<docstring>
\"\"\"
Checks if object1 has the same size as object2
Args:
   image (IMAGE): Image that the object is contained in.
   x_1 (int): X coordinate of object 1 in pixel space.
   y_1 (int): Y coordinate of object 1 in pixel space.
   x_2 (int): X coordinate of object 2 in pixel space.
   y_2 (int): Y coordinate of object 2 in pixel space.
Returns:
   bool: True if object 1 has the same size as object 2, False otherwise
\"\"\"
</docstring>
<signature>def same_size(image, x_1, y_1, x_2, y_2):</signature>
<implementation>
object_1_size = vqa(image=image, question=’What size is this object?’, x=x_1, y=y_1)
object_2_size = vqa(image=image, question=’What size is this object?’, x=x_2, y=y_2)
return object_1_size == object_2_size
</implementation>

Here are some helpful tips:
1) When you need to search over objects satisfying a condition, remember to check all the objects that satisfy the condition and don’t just return the first one.
2) You already have an initialized variable named "image" - no need to initialize it yourself!
3) When searching for objects to compare to a reference object, make sure to remove the reference object from the retrieved objects. You can check if two objects are
 the same with the same_object method.
Do not define new methods here, simply solve the problem using the existing methods.
Now, given the following docstring and signature, implement the method, using the API specification as necessary. Output the implementation inside <implementation></
implementation>.
Again, Output the implementation inside <implementation></implementation>.
<docstring>{docstring}</docstring>
<signature>{signature}</signature>
```

Figure 13. **Implementation Agent Prompt for CLEVR.** This prompt differs from the prompt used for OMNI3D-BENCH as we omit examples illustrating usage of the `get_2D_object_size` method. The prompt features *Weak ICL* examples illustrating correct usage of the pre-defined modules, as well as *Pseudo ICL* in the form of natural language instructions.

```
Implement a method given a docstring and method signature, using the API specification as necessary.
Current API:
{predef_signatures}
{generated_signatures}
Here are some examples of how to implement a method given its docstring and signature:
<docstring>
\"\"\" Locates objects that are on the left of the reference object.
Args:
   image (IMAGE): Image to search.
   ref_x (int): X coordinate of reference object in pixel space.
   ref_y (int): Y coordinate of reference object in pixel space.
Returns:
   points (list): list of [x, y] coordinates for objects in pixel space matching description to the left.
\"\"\"
</docstring>
<signature>def objects_left(image, ref_x, ref_y):</signature><implementation>
objects_left = []
all_objects = loc(image, object_prompt=’objects’)
for object_point in all_objects:
   x, y = object_point
   if same_object(image, ref_x, ref_y, x, y):
      continue
   if x < ref_x:
      objects_left.append(object_point)
return objects_left </implementation>
<docstring>
\"\"\" Gets the material of the given object.
Args:
   image (IMAGE): Image that the object is contained in.
   ref_x (int): X coordinate of reference object in pixel space.
   ref_y (int): Y coordinate of reference object in pixel space.
Returns:
   str: Material of the object.
\"\"\"
</docstring>
<signature>def object_material(image, ref_x, ref_y):</signature><implementation>
return vqa(image=image, question=’What material is this object?’, x=ref_x, y=ref_y) </implementation>
<docstring>
\"\"\" Checks if an object 1 is in front of object 2.
Args:
   image (IMAGE): Image that the object is contained in.
   x_1 (int): X coordinate of object 1 in pixel space.
   y_1 (int): Y coordinate of object 1 in pixel space.
   x_2 (int): X coordinate of object 2 in pixel space.
   y_2 (int): Y coordinate of object 2 in pixel space.
Returns:
   bool: True if object 1 is in front of object 2, False otherwise
\"\"\"
</docstring>
<signature>def in_front_of(image, x_1, y_1, x_2, y_2):</signature> <implementation>
depth_1, depth_2 = depth(image, x_1, y_1), depth(image, x_2, y_2)
return depth_1 < depth_2 </implementation>
<docstring>
\"\"\" Checks if object1 has the same size as object2
Args:
   image (IMAGE): Image that the object is contained in.
   x_1 (int): X coordinate of object 1 in pixel space.
   y_1 (int): Y coordinate of object 1 in pixel space.
   x_2 (int): X coordinate of object 2 in pixel space.
   y_2 (int): Y coordinate of object 2 in pixel space.
   epsilon (float): Acceptable margin of error in sizes.
Returns:
   bool: True if object 1 has the same size as object 2, False otherwise
\"\"\"
</docstring>
<signature>def same_size(image, x_1, y_1, x_2, y_2, epsilon):</signature> <implementation>
object_1_height, object_1_width = get_2D_object_size(image, x_1, y_1)
object_2_height, object_2_width = get_2D_object_size(image, x_2, y_2)
return abs(object_1_height - object_2_height) < epsilon and abs(object_1_width - object_2_width) < epsilon </implementation>
<docstring>
\"\"\" Returns a list of objects in the images
Args:
   image (IMAGE): Image to search for objects in
Returns:
   list: List of strings corresponding to all of the objects in the image.
\"\"\"
</docstring>
<signature>def get_object_list(image):</signature> <implementation>
objects = []
object_points = loc(image, object_prompt=’objects’)
for object_point in object_coords:
   obj_x, obj_y = object_point
   objects.append(vqa(image, "What is this object?", obj_x, obj_y))
return objects </implementation>
Here are some helpful definitions:
1) 2D distance/size refers to distance/size in pixel space. 2) 3D distance/size refers to distance/size in the real world. 3D size is equal to 2D size times the
depth of the object. 3) "On" is defined as the closest object ABOVE another object. Only use this definition for "on". 4) "Next to" is defined as the closest object.
 5) Width is the same as length. 6) "Depth" measures distance from the camera in 3D.
Here are some helpful tips:
1) When you need to search over objects satisfying a condition, remember to check all the objects that satisfy the condition and don’t just return the first one. 2)
You already have an initialized variable named "image" - no need to initialize it yourself! 3) When searching for objects to compare to a reference object, make sure
 to remove the reference object from the retrieved objects. You can check if two objects are the same with the same_object method. 4) Do not assume that the objects
you see in these questions are all of the objects you will see, keep the methods general. 5) If two objects have the same 2D width, then the object with the largest
depth has the largest 3D width. 6) If two objects have the same 2D height, then the object with the largest depth has the largest 3D height. 7) 2D sizes convey the
height and width in IMAGE SPACE. To convert to height and width in 3D space, it needs to be multiplied by the depth! 8) If you are given a reference size, scale your
 output predicted size accordingly! Do not define new methods here, simply solve the problem using the existing methods. Now, given the following docstring and
signature, implement the method, using the API specification as necessary. Output the implementation inside <implementation></implementation>. Again, Output the
implementation inside <implementation></implementation>.
<docstring>
{docstring}
</docstring>
<signature>{signature}</signature>
```

Figure 14. **Implementation Agent Prompt for OMNI3D-BENCH.** The prompt features *Weak ICL* examples illustrating correct usage of the pre-defined modules, as well as *Pseudo ICL* in the form of natural language instructions and definitions.

```
You are an expert logician capable of answering spatial reasoning problems with code. You excel at using a predefined
API to break down a difficult question into simpler parts to write a program that answers spatial and complex
reasoning problem.
Answer the following question using a program that utilizes the API to decompose more complicated tasks and solve the
problem.
Available sizes are {{small, large}}, available shapes are {{square, sphere, cylinder}}, available material types are
{{rubber, metal}}, available colors are {{gray, blue, brown, yellow, red, green, purple, cyan}}.
The question may feature attributes that are outside of the available ones I specified above. If that's the case,
please replace them to the most appropriate one from the attributes above.
I am going to give you an example of how you might approach a problem in psuedocode, then I will give you an API and
some instructions for you to answer in real code.

Example:
Question: "What is the shape of the matte object in front of the red cylinder?"
Solution:
1) Find all the cylinders (loc(image, 'cylinders'))
2) If cylinders are found, loop through each of the cylinders found
3) For each cylinder found, check if the color of this cylinder is red. Store the red cylinder if you find it and
break from the loop.
4) Find all the objects.
5) For each object, check if the object is rubber (matte is not in the available attributes, so we replace it with
rubber)
6) For each rubber object O you found, check if the depth of O is less than the depth of the red cylinder
7) If that is true, return the shape of that object

Now here is an API of methods, you will want to solve the problem in a logical and sequential manner as I showed you
------------------ API ------------------
{pre_defined_signatures}
{api}
------------------ API ------------------
Please do not use synonyms, even if they are present in the question.
Using the provided API, output a program inside the tags <program></program> to answer the question.
It is critical that the final answer is stored in a variable called "final_result".
Ensure that the answer is either yes/no, one word, or one number.
Here are some helpful tips:
1) When you need to search over objects satisfying a condition, remember to check all the objects that satisfy the
condition and don't just return the first one.
2) You already have an initialized variable named "image" - no need to initialize it yourself! 3) Do not define new
methods here, simply solve the problem using the existing methods.
3) When searching for objects to compare to a reference object, make sure to remove the reference object from the
retrieved objects. You can check if two objects are the same with the same_object method.
Again, available sizes are {{small, large}}, available shapes are {{square, sphere, cylinder}}, available material
types are {{rubber, metal}}, available colors are {{gray, blue, brown, yellow, red, green, purple, cyan}}.
Again, answer the question by using the provided API to write a program in the tags <program></program> and ensure the
 program stores the answer in a variable called "final_result".
It is critical that the final answer is stored in a variable called "final_result".
Ensure that the answer is either yes/no, one word, or one number.
AGAIN, answer the question by using the provided API to write a program in the tags <program></program> and ensure the
 program stores the answer in a variable called "final_result".
You do not need to define a function to answer the question - just write your program in the tags. Assume "image" has
already been initialized - do not modify it!
<question>{question}</question>
```

Figure 15. **Program Agent Prompt for CLEVR.** In the prompt, we provide a list of all available attributes in CLEVR, a *Pseudo ICL* example in natural language, and some helpful tips.

```
You are an expert logician capable of answering spatial reasoning problems with code. You excel at using a predefined
API to break down a difficult question into simpler parts to write a program that answers spatial and complex
reasoning problem.
Answer the following question using a program that utilizes the API to decompose more complicated tasks and solve the
problem.
I am going to give you two examples of how you might approach a problem in psuedocode, then I will give you an API and
 some instructions for you to answer in real code.

Example 1:
Question: "What is the shape of the red object in front of the blue pillow?"
Solution:
1) Find all the pillows (loc(image, 'pillow')).
2) If pillows are found, loop through each of the pillows found.
3) For each pillow found, check if the color of this pillow is blue. Store the blue pillow if you find it and break
from the loop.
4) Find all the objects.
5) For each object, check if the object is red.
6) For each red object O you found, check if the depth of O is less than the depth of the blue pillow.
7) If that is true, return the shape of that object.

Example 2:
Question: "How many objects have the same color as the metal bowl?"
Solution:
1) Set a counter to 0
2) Find all the bowls (loc(image, 'bowls')).
3) If bowls are found, loop through each of the bowls found.
4) For each bowl found, check if the material of this bowl is metal. Store the metal bowl if you find it and break
from the loop.
5) Find and store the color of the metal bowl.
6) Find all the objects.
7) For each object O, check if O is the same object as the small bowl (same_object(image, metal_bowl_x, metal_bowl_y,
object_x, object_y)). If it is, skip it.
8) For each O you don't skip, check if the color of O is the same as the color of the metal bowl.
9) If it is, increment the counter.
10) When you are done looping, return the counter.

Now here is an API of methods, you will want to solve the problem in a logical and sequential manner as I showed you
------------------ API ------------------
{predef_signatures}
{api}
------------------ API ------------------
Please do not use synonyms, even if they are present in the question.
Using the provided API, output a program inside the tags <program></program> to answer the question.
It is critical that the final answer is stored in a variable called "final_result".
Ensure that the answer is either yes/no, one word, or one number.
Here are some helpful definitions:
1) 2D distance/size refers to distance/size in pixel space.
2) 3D distance/size refers to distance/size in the real world. 3D size is equal to 2D size times the depth of the
object.
3) "On" is defined as the closest object ABOVE another object. Only use this definition for "on".
4) "Next to" is defined as the closest object.
5) Width is the same as length.
6) "Depth" measures distance from the camera in 3D.
Here are some helpful tips:
1) When you need to search over objects satisfying a condition, remember to check all the objects that satisfy the
condition and don't just return the first one.
2) You already have an initialized variable named "image" - no need to initialize it yourself!
3) When searching for objects to compare to a reference object, make sure to remove the reference object from the
retrieved objects. You can check if two objects are the same with the same_object method.
4) Do not assume that the objects you see in these questions rae all of the objects you will see, keep the methods
general.
5) If two objects have the same 2D width, then the object with the largest depth has the largest 3D width.
6) If two objects have the same 2D height, then the object with the largest depth has the largest 3D height.
7) 2D sizes convey the height and width in IMAGE SPACE. To convert to height and width in 3D space, it needs to be
multiplied by the depth!
8) If you are given a reference size, scale your output predicted size accordingly!
Again, answer the question by using the provided API to write a program in the tags <program></program> and ensure the
 program stores the answer in a variable called "final_result".
It is critical that the final answer is stored in a variable called "final_result".
Ensure that the answer is either yes/no, one word, or one number.
AGAIN, answer the question by using the provided API to write a program in the tags <program></program> and ensure the
 program stores the answer in a variable called "final_result".
You do not need to define a function to answer the question - just write your program in the tags. Assume "image" has
already been initialized - do not modify it!
<question>{question}</question>
```

Figure 16. **Program Agent Prompt for OMNI3D-BENCH.** The prompt features *Pseudo ICL* in the form of two natural language examples and helpful tips for handling 2D and 3D dimensions.