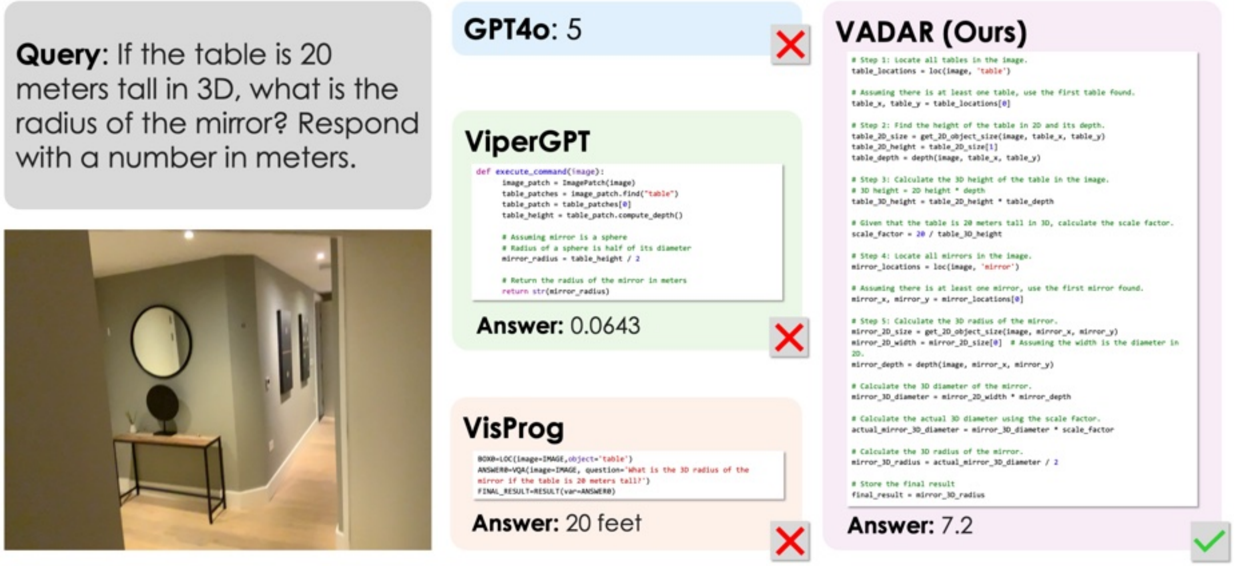
Visual Agentic AI for Spatial Reasoning with a Dynamic API

Damiano Marsili* Rohun Agrawal* Yisong Yue Georgia Gkioxari

California Institute of Technology



Query: If the table is 20 meters tall in 3D, what is the radius of the mirror? Respond with a number in meters.

```python
def execute_command(image):
    image_patch = ImagePatch(image)
    table_patches = image_patch.find("table")
    table_patch = table_patches[0]
    table_height = table_patch.compute_depth()

    # Assuming mirror is a sphere
    # Radius of a sphere is half of its diameter
    mirror_radius = table_height / 2

    # Return the radius of the mirror in meters
    return str(mirror_radius)
```

```
BOX0=LOC(image=IMAGE,object='table')
ANSWER0=VQA(image=IMAGE, question='What is the 3D radius of the
mirror if the table is 20 meters tall?')
FINAL_RESULT=RESULT(var=ANSWER0)
```

```python
# Step 1: Locate all tables in the image.
table_locations = loc(image, 'table')

# Assuming there is at least one table, use the first table found.
table_x, table_y = table_locations[0]

# Step 2: Find the height of the table in 2D and its depth.
table_2D_size = get_2D_object_size(image, table_x, table_y)
table_2D_height = table_2D_size[1]
table_depth = depth(image, table_x, table_y)

# Step 3: Calculate the 3D height of the table in the image.
# 3D height = 2D height * depth
table_3D_height = table_2D_height * table_depth

# Given that the table is 20 meters tall in 3D, calculate the scale factor.
scale_factor = 20 / table_3D_height

# Step 4: Locate all mirrors in the image.
mirror_locations = loc(image, 'mirror')

# Assuming there is at least one mirror, use the first mirror found.
mirror_x, mirror_y = mirror_locations[0]

# Step 5: Calculate the 3D radius of the mirror.
mirror_2D_size = get_2D_object_size(image, mirror_x, mirror_y)
mirror_2D_width = mirror_2D_size[0]  # Assuming the width is the diameter in
2D.
mirror_depth = depth(image, mirror_x, mirror_y)

# Calculate the 3D diameter of the mirror.
mirror_3D_diameter = mirror_2D_width * mirror_depth

# Calculate the actual 3D diameter using the scale factor.
actual_mirror_3D_diameter = mirror_3D_diameter * scale_factor

# Calculate the 3D radius of the mirror.
mirror_3D_radius = actual_mirror_3D_diameter / 2

# Store the final result
final_result = mirror_3D_radius
```

Figure 1. Spatial reasoning in 3D is challenging as it requires multiple steps of grounding and inference. We introduce a benchmark for 3D understanding with complex queries; an example is shown here. To tackle these queries we propose a training-free agentic approach, VADAR, that dynamically generates new skills in Python and thus can handle a wider range of queries compared to prior methods.

Abstract

Visual reasoning – the ability to interpret the visual world – is crucial for embodied agents that operate within three-dimensional scenes. Progress in AI has led to vision and language models capable of answering questions from images. However, their performance declines when tasked with 3D spatial reasoning. To tackle the complexity of such reasoning problems, we introduce an agentic program synthesis approach where LLM agents collaboratively generate a Pythonic API with new functions to solve common subproblems. Our method overcomes limitations of prior approaches that rely on a static, human-defined API, allowing it to handle a wider range of queries. To assess AI capabilities for 3D understanding, we introduce a new benchmark of queries involving multiple steps of grounding and inference. We show that our method outperforms prior zero-shot models for visual reasoning in 3D and empirically validate the effectiveness of our agentic framework for 3D spatial reasoning tasks. Project website: https://glab-caltech.github.io/vadar/

1. Introduction

Consider Fig. 1. Here, a person or an agent wants to determine the radius of the mirror in the image, given that the table is 20 meters tall. Answering this question requires visual reasoning, a crucial step toward achieving general-purpose AI. Visual reasoning enables machines to analyze and make sense of the visual world. Humans rely heavily on visual cues to navigate complex environments, interact with objects and make informed decisions. Our goal is to build intelligent agents that can do the same. Recent advances in AI have produced vision and language models (VLMs) [1, 2, 8, 36] that can answer questions from images. Although impressive, these models excel primarily at category-level semantic understanding. Their performance significantly declines when tasked with spatial understanding within the three-dimensional world [6, 19, 38].

Returning to Fig. 1, to answer the query, an AI agent must first locate the relevant objects, determine their dimensions in pixel space, use their depth to calculate their 3D sizes, and finally compute the mirror's radius using the table's height. This is a complex sequence of tasks, involving multiple steps of understanding, grounding, and infer-

[*] Equal contribution.

ence. GPT4o [1], a state-of-the-art VLM trained on extensive datasets, gives a wrong final answer.

To address the complexity of 3D spatial reasoning tasks, we propose a system of agents working collaboratively to create executable programs for a given image. Our approach leverages LLM agents that *dynamically* define and expand a domain-specific language (DSL) *as needed*, generating new functions, skills and reasoning, in two phases: the **API Generation** and the **Program Synthesis** stage. Vision specialists – an object detector, a depth estimator and object attribute predictor – help the agents execute the program. We name our approach VADAR, as it integrates Visual, Agentic, Dynamic AI for Reasoning. VADAR belongs in the family of visual program synthesis methods, like ViperGPT [35] and VisProg [12], but addresses a key limitation in these approaches: their reliance on a static, human-defined DSL, which restricts them to a predefined range of functionality. This limitation is evident in Fig. 1, where ViperGPT generates an incomplete, inaccurate program and VisProg defaults to a holistic visual question answer (VQA) approach for answering the query. VADAR's output in Fig. 1 demonstrates its ability to tackle a wider range of visual queries.

We evaluate 3D spatial reasoning using challenging benchmarks designed for rigorous assessment of 3D understanding. Our evaluation includes CLEVR [18] and our newly introduced benchmark, OMNI3D-BENCH, based on Omni3D [5]; Fig. 1 shows an example. Both datasets emphasize visual queries involving relative depth, size, and object location, often conditioned on measurement hypotheses, requiring grounding and 3D inference. This contrasts with previous spatial reasoning benchmarks like GQA [16], which primarily emphasize appearance-based reasoning.

At a high level, VADAR roughly mirrors the workflow of a software engineer when defining, implementing, and testing new software solutions for a given problem. Leveraging its agentic design, VADAR autonomously defines and implements functions such as `_find_closest_object_3D`, `_is_behind`, `_count_objects_by_attributes_and_position`, `_is_left_of`, and more. These functions are used by the Program Agent, resulting in more concise programs, less output tokens and thus a lower likelihood of errors from LLM-generated predictions. We empirically show that VADAR outperforms a *no-API* agent by 6%, highlighting the value of general, reusable, functions within an API. Moreover, we show that our generated API significantly surpasses a static, human-defined API used in [12, 35], by more than 20% on CLEVR. VADAR performs competitively with state-of-the-art VLMs, on OMNI3D-BENCH, while also providing executable programs.

Considering the rapid progress in AI, one might wonder if methods like VADAR can dominate monolithic VLMs in 3D spatial reasoning. One clear advantage of VADAR is its ability to generate interpretable programs. However, our experiments highlight another key potential. Improving VLMs for 3D reasoning would require extensive datasets of image-question-answer tuples with 3D information, an onerous endeavor. In contrast, our experiments show that if the component vision models – an object detector, an attribute predictor and depth estimator – were replaced with oracle versions, VADAR would achieve 83.0% accuracy, 24% higher from the best VLM. This indicates that VADAR is bottlenecked by the performance of its vision specialists. Thus, an alternative path to scaling 3D spatial reasoning could be through improving specialized vision models, which tackle a simpler problem than general-purpose VQA and for which training data is more readily available.

2. Related Work

Our work draws from areas of language modeling, visual program synthesis and library learning.

VLMs for Spatial Reasoning. LLMs [1, 2, 9, 36] are trained on large corpora of text, including domain specific languages (DSLs) such as Python. Their multi-modal variants incorporate images and are additionally trained on image-text pairs showing impressive results for visual captioning and vision question-answering (VQA) [3]. Despite their strong performance, their ability to reason beyond category-level semantic queries is limited. Recent work [19, 38] shows that VLMs suffer on visual tasks such as grounding spatial relationships and inferring object-centric attributes. SpatialRGPT [7] and SpatialVLM [6] use data synthesis pipelines to generate templated queries for spatial understanding. We compare to SpatialVLM and show that it struggles to tackle 3D spatial reasoning queries.

Visual Program Synthesis. Recent advances in visual reasoning have led to methods which improve upon the capabilities of vision-based models by composing them symbolically via program synthesis. VisProg [12] prompts an LLM to generate an executable program of a specified DSL that calls and combines vision specialists – OwlViT [29] for object detection, CLIP [32] for classification, and ViLT [21] for VQA. ViperGPT [35] directly generates Python code by providing a Python API specification to the LLM agent and adds MiDaS [33] as the vision specialist for depth estimation, in addition to GLIP [25] and X-VLM [45] for vision-language tasks. Both approaches rely on a predefined DSL, which narrows the scope of applicability and makes these methods difficult to extend to a wider range of queries. Similar to ViperGPT, we use Python as the interface for our LLM agents, but we don't define the API a-priori. We instead rely on our agentic workflow to generate the API needed to tackle complex spatial reasoning queries. We compare to ViperGPT and VisProg and show that both struggle to generate accurate programs for complex queries, often completely ignoring part of the query.

Library Learning. An emerging field in LLM research focuses on the dynamic creation and extension of a set of reusable functions during problem-solving. Early work on library learning predates the use of LLMs [10, 23, 39], and focuses on a common architecture of iteratively proposing new programs and synthesizing commonly used components into a library. Modern approaches follow this same paradigm, but use LLMs to accelerate the synthesis of useful programs, applied to gaming [40], 3D graphics scripting [15], theorem proving [37], and symbolic regression [11].

Neuro-symbolic AI generates interpretable symbolic components for complex tasks and has been explored for a wide range of fields, including spatial reasoning [28], grounding of 3D point clouds [13], mechanistic modeling in scientific domains [11, 34], logical reasoning [30], amongst other areas. Closer to us is the logic-enhanced LLM, LEFT [14], that uses a dynamic DSL of first order logic structures and differentiably executes them using domain-specific modules. These modules, instantiated as MLPs, ground spatial concepts, *e.g.* *"is left of"*, and are *trained with supervision*. On CLEVR, VADAR, which is *training-free*, achieves the same performance as LEFT when trained with $\geq 10,000$ training samples. A benefit of our training-free approach is that it scales to new domains where 3D supervision is hard to acquire, as we show on our OMNI3D-BENCH.

Spatial Reasoning Benchmarks. Existing benchmarks test aspects of visual reasoning with free-form language [4, 24]. We focus on natural-image based ones. VQA [3] introduced the task of visual question answering. GQA [16] is a popular large-scale VQA benchmark with questions that pertain to object and attribute recognition, of mostly a single-step inference – *"What color is the cat next to the chair?"*, *"What type of vehicle is on top of the road?"*, *"Do the wildflowers look ugly?"*. RefCOCO [20] targets object localization with referring expressions such as *"the man in a red shirt"*. What's up [19] quantifies comprehension of basic 2D spatial relations such as *"left of"* and *"above"*. These benchmarks evaluate aspects of visual reasoning, but critically omit 3D understanding. Q-Spatial Bench [26] focuses solely on absolute 3D measurements. Cambrian-1 [38] proposes a VQA benchmark repurposing images and annotations from Omni3D [5], but its queries focus on the relative depth and depth ordering of objects with (2 or 3)-choice questions. Our benchmark also repurposes Omni3D annotations, but in contrast to Cambrian-1, we design more complex queries that extend beyond depth ordering and multiple choice. Concurrent to our work, VSI-Bench [44] introduces a video understanding benchmark focused on spatial relationships, which we discuss extensively in Appendix D.

3. Method

At the core of our approach is a dynamic API generated by LLMs that can be extended to address new queries that require novel skills. The goal of the API is to break down complex reasoning problems into simpler subproblems with general modules that can be used during program synthesis. Our approach consists of an API Generation stage and a Program Synthesis stage, illustrated in Fig. 2.

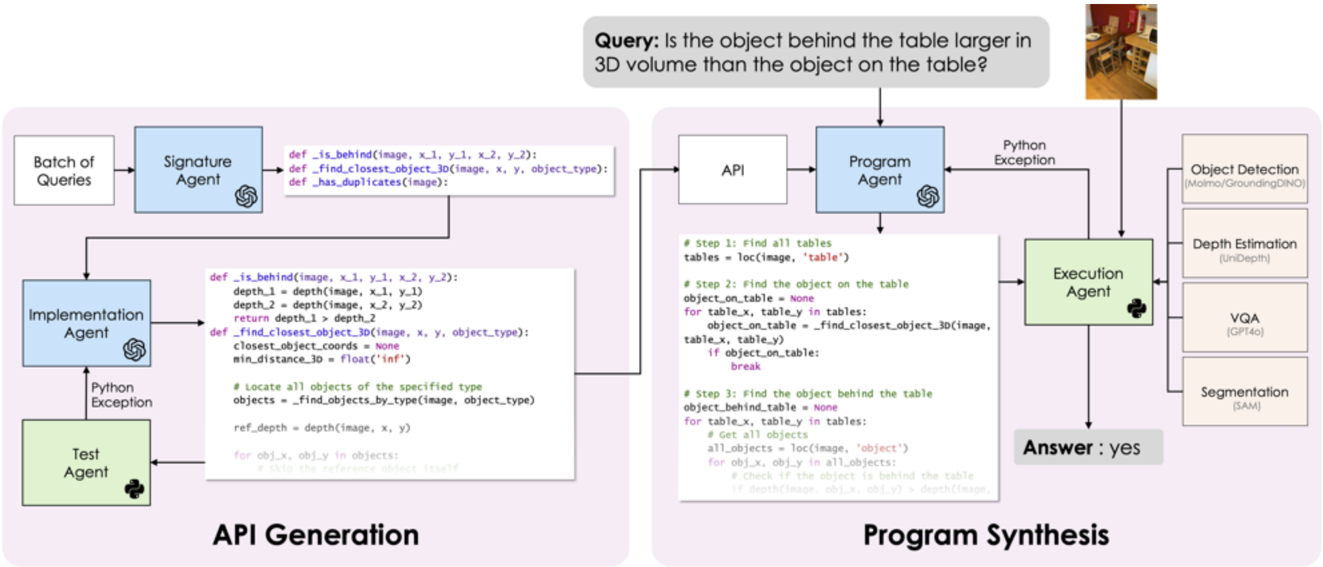
Vision Specialists. During program execution on the image, we employ vision models for solving visual subtasks: Molmo's [8] pointing model and GroundingDINO [27] are used to localize objects prompted with text (`loc`), SAM [22] returns the bounding box from the object's mask prompted with Molmo's points (`get_2D_object_size`), UniDepth [31] estimates the depth at an image location (`depth`), GPT4o is utilized as a VQA module to query object attributes (color, material) from an image with the target object bounding box overlayed (`vqa`). We initialize the API with these functions. The API also includes `same_object` that computes the overlap of two object bounding boxes to determine if the objects are the same.

3.1. API Generation

Algorithm 1: VADAR: API Generation

```
Data: Questions $\mathcal{Q}$
$\mathcal{S} \leftarrow \{\}$                          // Signatures
$\mathcal{A} \leftarrow \{$Vision Models$\}$        // API Methods
for batch $B \subset \mathcal{Q}$ do
    $\mathcal{S} \leftarrow \mathcal{S} \cup$ SignatureAgent($B$)
end
for $S \in \mathcal{S}$ do
    $e_S \leftarrow 0$                         // Error count
    $A \leftarrow$ ImplementationAgent($S$)
    $E \leftarrow$ TestAgent($A$)
    if Python Exception $E$ then
        if $e_S = 5$ then continue
        else if $E$ is "undefined method $U$" then
            $e_S \leftarrow e_S + 1$
            Recursively implement $U$
        else
            $e_S \leftarrow e_S + 1$
            Re-implement $S$ using $E$
        end
    else
        $\mathcal{A} \leftarrow \mathcal{A} \cup A$
    end
end
return $\mathcal{A}$
```

Algorithm 1 describes the API Generation. Here, the **Signature Agent** and the **Implementation Agent** collaborate to define and implement new functions *as needed* to



```python
def _is_behind(image, x_1, y_1, x_2, y_2):
def _find_closest_object_3D(image, x, y, object_type):
def _has_duplicates(image):
```

```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
def _find_closest_object_3D(image, x, y, object_type):
    closest_object_coords = None
    min_distance_3D = float('inf')

    # Locate all objects of the specified type
    objects = _find_objects_by_type(image, object_type)

    ref_depth = depth(image, x, y)

    for obj_x, obj_y in objects:
        # Skip the reference object itself
```

```python
# Step 1: Find all tables
tables = loc(image, 'table')

# Step 2: Find the object on the table
object_on_table = None
for table_x, table_y in tables:
    object_on_table = _find_closest_object_3D(image,
table_x, table_y)
    if object_on_table:
        break

# Step 3: Find the object behind the table
object_behind_table = None
for table_x, table_y in tables:
    # Get all objects
    all_objects = loc(image, 'object')
    for obj_x, obj_y in all_objects:
        # Check if the object is behind the table
        if depth(image, obj_x, obj_y) > depth(image,
```

Figure 2. **Overview.** VADAR consists of an API generation stage and a program synthesis stage. The Signature & Implementation Agents generate an API that is used by the Program Agent to produce a program to answer the question, executed by the Execution Agent.

aid in solving the queries. First, the Signature Agent receives a batch of N queries ($N = 15$), *without answers*, and is instructed to produce general method signatures for subproblems that could arise when answering those kinds of queries. The Implementation Agent then implements the signatures in Python. Examples of signatures and their implementations are shown in Fig. 2.

Prompting the Signature Agent. The agent receives the current API state as docstrings so it avoids duplicating existing methods. We observed that our Signature Agent performed better without in-context examples as it produced a more diverse API with wider potential functionality.

Prompting the Implementation Agent. The Implementation Agent receives all other signatures in the API along with the signature it needs to implement, so it can use other API methods in its implementation, enabling a hierarchy in the API. In contrast to the Signature Agent, providing in-context examples significantly enhances the Implementation Agent's output, as implementation prioritizes accuracy over diversity. We refer to these examples as *weak* in-context learning (ICL), as they guide correct method implementation in Python, unlike *strong* ICL, which breaks down queries into full programs. Prompts for both agents and weak-ICL examples are found in the Appendix.

Depth-First Implementation. Once a method is implemented from its signature, the Test Agent, a Python interpreter, runs it using placeholder inputs. If a runtime error occurs, the Test Agent signals the Implementation Agent to revise it with the exception message. However, if the implementation relies on another yet-to-be-implemented API method, the test run cannot proceed. In this case, the Implementation Agent traverses an implicit dependency graph, depth-first, ensuring that prerequisite methods are implemented first (see Algo. 1).

Consider the following example where the signatures `get_color`, `find_objects_by_color`, `count_objects_left_of`, and `is_left_of`, are defined by the Signature Agent, in that order. First, the Implementation Agent will implement `get_color`, the Test Agent will be called, and barring no runtime errors, the method will be complete. Then, the implementation for `find_objects_by_color` uses `get_color`, which is implemented, so the Test Agent only checks for Python errors. If `count_objects_left_of` attempts to use `is_left_of`, the Test Agent will detect that `is_left_of` is not implemented and recursively call the Implementation Agent to implement `is_left_of`, followed by `count_objects_left_of`.

In the event a cycle in the dependency graph is persistent after attempting the implementation of those methods 5 times, the methods in the cycle are deleted. Empirically, we rarely detect such cycles, which can be attributed to the Signature Agent producing multiple signatures at once, tending to avoid proposing signatures that overlap in function.

3.2. Program Synthesis

The **Program Agent** receives the generated API and a single question as input. Its task is to generate Python code that leverages the API to solve the question. The Execution Agent, another Python interpreter, executes the program line-by-line. In the event of a Python error, it provides the Program Agent with the exception, and a new program is generated. This is repeated at most 5 times, after which the program returns an execution error.

Algorithm 2: VADAR: Program Synthesis

```
Data: Image-Query pairs $\mathcal{D} = \{(I, Q)\}$,
      API methods $\mathcal{A}$
$\mathcal{R} \leftarrow \{\}$                         // Results
for $(I, Q) \in \mathcal{D}$ do
    $e_P \leftarrow 0$                    // Error count
    $P \leftarrow$ ProgramAgent$(Q, \mathcal{A})$
    $E, R \leftarrow$
     ExecutionAgent$(P, I,$ Vision Models$)$
    if Python Exception $E$ and $e_P < 5$ then
        $e_P \leftarrow e_P + 1$
        Re-generate $P$ using $E$
    else
        $\mathcal{R} \leftarrow \mathcal{R} \cup R$
    end
end
return $\mathcal{R}$
```

Prompting the Program Agent. Following the success of Chain-of-Thought (CoT) prompting [41], we instruct the Program Agent to create a plan before generating the corresponding program. In-context examples boost the Program Agent's performance. However, unlike VisProg [12] and ViperGPT [35] that use strong-ICL, we use API-agnostic natural language instructions since the API is not predefined, making it impossible to provide full program examples. These instructions help for the same reason as with the Implementation Agent, to focus on correctness. The prompt for the Program Agent is provided in the Appendix.

Test & Execution Agent vs Critics. In modern library learning, LLM agents, or critics, evaluate the quality and utility of learned functions. Our Test and Execution Agents also assess method quality, but we opt for deterministic critics that leverage the full Python runtime, signaling LLM Agents with Python exceptions in case of errors.

4. Experiments

We evaluate our approach on challenging spatial reasoning benchmarks, demonstrating that a dynamically generated API outperforms the static, human-defined APIs in ViperGPT [35] and VisProg [12] by a large margin. Additionally, we compare against state-of-the-art monolithic VLMs trained on billions of (image, question, answer) samples, showing that our method competes favorably and even surpasses them on certain question types while offering interpretable reasoning steps for complex queries.

4.1. A Benchmark for Spatial Reasoning in 3D

We evaluate 3D spatial reasoning using CLEVR, and our newly introduced benchmark, OMNI3D-BENCH.

CLEVR [18] consists of (image, question, answer) tuples. Each image contains 2-10 objects of 3 different shapes, 8 colors, 2 materials, and 2 sizes. Despite the simplicity of the scenes, the questions in CLEVR are complex, *e.g.*, *"There is a large ball right of the large metal sphere that is left of the large object that is behind the small brown sphere; what color is it?"*. Our CLEVR benchmark contains 1,155 samples, 400 of which require a numerical answer, 399 are yes/no questions, and 356 are multiple-choice questions.

OMNI3D-BENCH is sourced from Omni3D [5], a dataset of images from diverse real-world scenes with 3D object annotations. We repurpose images from Omni3D to a VQA benchmark, with questions about 3D information portrayed in the image, such as *"If the height of the front most chair is 6 meters in 3D, what is the height in 3D of the table in the image?"* and *"How many bottles would you have to stack on top of each other to make a structure as tall in 3D as the armchair?"*. OMNI3D-BENCH complements CLEVR with *non-templated* queries pertaining to 3D locations and sizes of objects. Our queries test 3D reasoning, as they require grounding objects in 3D and combining predicted attributes to reason about distances and dimensions in three dimensions. OMNI3D-BENCH consists of 500 extremely challenging (image, question, answer) tuples.

We compare our proposed benchmark to GQA [16], a popular visual reasoning dataset. GQA derives queries from scene graphs which primarily pertain to the visual appearance and attributes of objects. Example queries in GQA are *"Is there a red truck or bus?"*, *"Is the field short and brown?"* and *"Is the chair in the top part of the image?"*. These are significantly simpler to queries in CLEVR and OMNI3D-BENCH which involve multiple steps of grounding and inference in two- and three- dimensions.

4.2. Results on Spatial Reasoning in 3D

Tab. 1 compares our approach, VADAR, to state-of-the-art VLMs and Program Synthesis methods. Fig. 3 additionally compares to the neuro-symbolic LEFT [14]. VADAR uses GPT4o with a temperature of 0.7 for all agents.

VLMs vs VADAR. VLMs, such as GPT4o [1], Claude-Sonnet [2], Gemini [36], Llama3.2-11B [9], and Molmo-7B [8], are monolithic models trained on vast image-question-answer datasets, likely including samples with spatial and 3D information. We expect them to perform well on related tasks. We also compare to SpaceMantis [6, 17], the most recent and largest SpatialVLM [6] variant, finetuned on data with 3D information. We analyze performance based on three answer types: yes/no, multiple-choice, and numerical answers. For queries with floating point answers, we report MRA [44] with thresholds $\mathcal{C} = \{0.5, 0.55, ..., 0.95\}$ for outputs $\hat{y}$ and ground truth y: $\mathcal{MRA} = \frac{1}{|\mathcal{C}|} \sum_{\theta \in \mathcal{C}} \mathbb{1}\left(\frac{|\hat{y} - y|}{y} < 1 - \theta \right)$

		CLEVR				OMNI3D-BENCH				
		numeric	y/n	multi-choice	Total	numeric (ct)	numeric (other)	y/n	multi-choice	Total
VLMs	GPT4o [1]	52.3	63.0	60.0	58.4	**28.1**	**35.5**	**66.7**	57.2	**42.9**
	Claude3.5-Sonnet [2]	44.7	61.4	**72.2**	**58.9**	22.4	20.6	62.2	50.6	32.2
	Llama3.2 [9]	34.6	45.6	49.0	42.8	24.3	19.3	47.5	27.4	25.6
	Gemini1.5-Pro [36]	44.9	59.7	67.0	56.9	25.2	28.1	46.2	37.6	32.0
	Gemini1.5-Flash [36]	43.1	58.8	56.8	52.8	24.3	27.6	51.1	52.9	35.0
	Molmo [8]	11.0	42.6	51.4	34.4	21.4	21.7	29.3	41.2	26.1
	SpaceMantis [6, 17]	14.5	52.9	32.3	33.2	20.0	21.7	50.6	48.2	30.3
Program Synthesis	ViperGPT [35]	20.5	43.4	13.4	26.2	20.0	15.4	56.0	42.4	26.7
	VisProg [12]	16.7	48.4	28.3	31.2	2.9	0.9	54.7	25.9	13.5
	VADAR (ours)	**53.3**	**65.3**	40.8	53.6	21.7	**35.5**	56.0	**57.6**	40.4

Table 1. **Accuracy (%) on CLEVR and OMNI3D-BENCH.** We compare to state-of-the-art monolithic VLMs and Program Synthesis approaches. For each benchmark, we breakdown performance for *numeric (ct)*, *numeric (other)*, *yes/no* and *multiple-choice* answers and report total accuracy. For *numeric (other)* queries, which require floating point answers, we report MRA. VADAR outpeforms ViperGPT and VisProg with a big margin. VADAR outperforms all large VLMs on OMNI3D-BENCH except GPT4o, which it is narrowly behind.

	CLEVR				OMNI3D-BENCH				
	numeric	y/n	multi-choice	Total	numeric (ct)	numeric (other)	y/n	multi-choice	Total
ViperGPT [35]	38.5	57.8	30.2	42.6	50.0	17.8	66.7	49.3	54.9
VisProg [12]	25.3	52.5	41.8	39.9	**100.0**	23.5	68.5	66.7	66.0
VADAR (ours)	**82.4**	**85.4**	**81.0**	**83.0**	**100.0**	**82.3**	**100.0**	**94.1**	**94.4**
GPT4o	52.3	63.0	66.0	58.4	30.0	29.4	77.8	44.0	53.7
Claude3.5-Sonnet	44.7	61.4	72.2	58.9	30.0	35.3	83.3	56.0	59.3

Table 2. **Oracle accuracy (%) on CLEVR and OMNI3D-BENCH.** We assess program synthesis correctness by replacing vision specialists with oracle variants. We report oracle accuracy on CLEVR and a smaller subset of OMNI3D-BENCH and compare to best performing monolithic VLMs on the same sets. VADAR's high oracle accuracy indicates its main limitation is the vision specialists' performance.

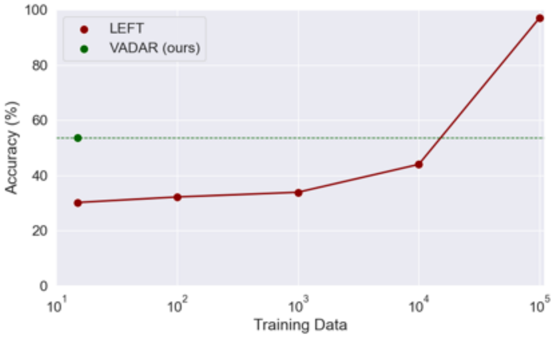


Figure 3. **LEFT [14] vs VADAR on CLEVR.** LEFT requires supervision. We vary the amount of training data (x-axis) and report accuracy (y-axis). VADAR requires *no* supervision but takes in 15 queries *without answers* to guide the creation of the API. VADAR outperforms LEFT trained with $\leq 10,000$ supervised examples.

From Tab. 1, we observe that on CLEVR, GPT4o, Claude-Sonnet, and Gemini perform best on average while VADAR slightly outperforms VLMs on numeric (by 1.0%) and yes/no answers (by 2.3%), while providing interpretable execution traces. On OMNI3D-BENCH, VADAR is behind GPT4o by just 2% and outperforms all other VLMs by more than 5%. Llama3.2-11B and Molmo-7B perform worse among VLMs likely due to their smaller size.

ViperGPT vs VisProg vs VADAR. VADAR outperforms both methods on both CLEVR and OMNI3D-BENCH by more than 20%. VisProg and VADAR use GPT4o as their LLM; ViperGPT uses GPT-3.5 as it performed better.

Separating program correctness from execution accuracy, Tab. 2 provides comparisons to ViperGPT and VisProg when vision specialists are replaced with oracle ones. On CLEVR, we use an Oracle Execution Agent that leverages the true scene annotations to provide the correct output automatically. For OMNI3D-BENCH, we use a smaller subset of 50 queries and manually verify program correctness as ground truth 3D information is not available for all objects in the scene. The results reveal that with oracle vision specialists, VADAR achieves an accuracy of 83.0% on CLEVR and 94.4% on OMNI3D-BENCH, compared to ViperGPT's 42.6% and 54.9%, and VisProg's 39.9% and 66.0% respectively. This suggests that VADAR supports a wider variety of queries, thanks to the dynamically generated API by our LLM agents, as opposed to the static, human-defined API in ViperGPT and VisProg. Our API allows for flexible integration of vision specialists, avoiding human biases – *e.g*., as in VisProg, where the pre-defined API guides the LLM to define "behind" by cropping the image above.

The high accuracy of VADAR with oracle vision specialists – more than 20% above Claude-Sonnet on CLEVR and more than 40% above GPT4o on OMNI3D-BENCH– suggests a promising path to scaling 3D spatial reasoning: improving specialized vision models. These models are easier to train than general-purpose VLMs, as they address simpler tasks with more accessible training data.

Fig. 4 shows programs generated by the methods. We

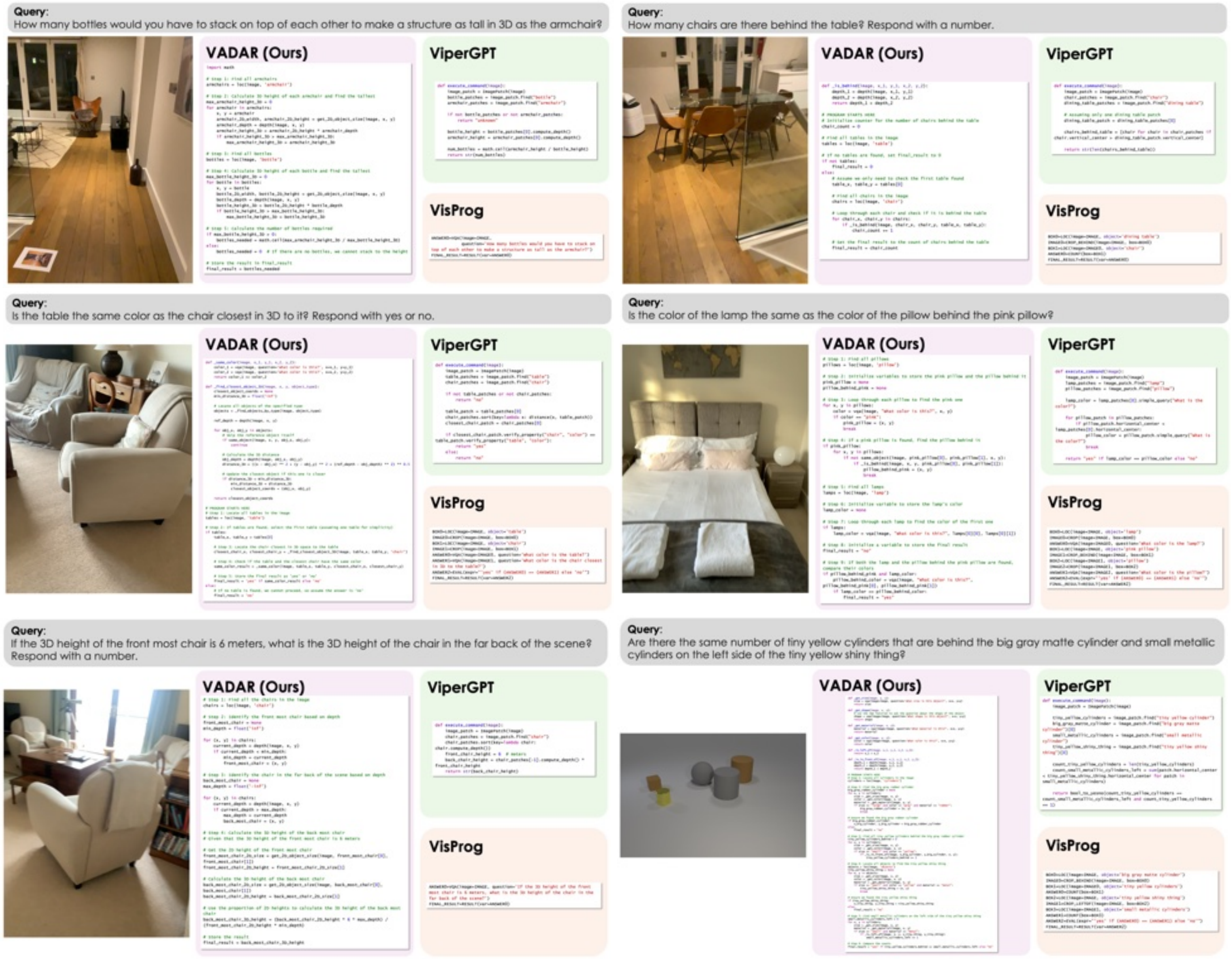

Query:
How many bottles would you have to stack on top of each other to make a structure as tall in 3D as the armchair?
VADAR (Ours)
ViperGPT
VisProg
Query:
How many chairs are there behind the table? Respond with a number.
VADAR (Ours)
ViperGPT
VisProg
Query:
Is the table the same color as the chair closest in 3D to it? Respond with yes or no.
VADAR (Ours)
ViperGPT
VisProg
Query:
Is the color of the lamp the same as the color of the pillow behind the pink pillow?
VADAR (Ours)
ViperGPT
VisProg
Query:
If the 3D height of the front most chair is 6 meters, what is the 3D height of the chair in the far back of the scene? Respond with a number.
VADAR (Ours)
ViperGPT
VisProg
Query:
Are there the same number of tiny yellow cylinders that are behind the big gray matte cylinder and small metallic cylinders on the left side of the tiny yellow shiny thing?
VADAR (Ours)
ViperGPT
VisProg

Figure 4. **Program outputs for VisProg, ViperGPT and VADAR.** For each example, we show the query, the input image, and the method's program generations. Queries are from our benchmark and pertain to 3D understanding of scenes. Zoom-in to read the programs.

observe that ViperGPT and VisProg tend to resort to direct VQA calls when questions are complex, as opposed to generating programs. In addition, ViperGPT often tends to produce incomplete programs, ignoring a significant portion of the query. Finally, both ViperGPT and VisProg often confuse above-behind and below-in front. This seems to be a semantic error for ViperGPT that uses a depth estimation module, like us, and a conceptual design error by VisProg that implements `CROP_BEHIND` to crop above in the image.

LEFT [14] vs VADAR. We also compare to the logic-enhanced neuro-symbolic approach LEFT [14], which uses trained modules to ground visual concepts in images, such as "`is left of`". Unlike LEFT, our approach is entirely training-free, while LEFT requires extensive supervision for module training. Fig. 3 reports the performance of LEFT on the CLEVR dataset when trained (to convergence) with varying training set sizes (x-axis). Although our approach does not require any explicit supervision, our API agent uses a small sample ($= 15$) of *questions only*, *without answers*, to construct the API. According to Fig. 3, we outperform LEFT trained with $\leq 10,000$ examples on CLEVR. Notably, it is not possible to evaluate LEFT on OMNI3D-BENCH due to its reliance on a large, domain-specific training set with appropriate 3D supervision, which is difficult to obtain for this benchmark or in general. This highlights an added advantage of our method: its ability to scale to new domains without the need for training.

Results on GQA. We report results on GQA [16], a widely used benchmark for spatial reasoning. As noted earlier, GQA queries emphasize object appearance and attributes, and primarily require one-step inference. Questions in GQA include *"What size is the doughnut the person is eating?"* and *"Who is sitting in front of the water?"*. Tab. 3 compares GPT4o, ViperGPT, VisProg, and VADAR. We observe different relative model performance compared to Tab. 1. Given the nature of GQA, it is not surprising that a monolithic and performant VLM like GPT4o would perform well, which our results confirm. Among the program

Method	GQA
GPT4o [1]	**54.9**
ViperGPT [35]	42.0
VisProg [12]	46.9
VADAR (ours)	46.1

Table 3. **Results on GQA** on a subset of testdev split. GQA focuses primarily on object appearance, not 3D spatial reasoning.

	CLEVR $_{100}$
No-API Agent	60.7
API Agent	64.0
+ Weak ICL	65.7
+ Pseudo ICL	66.7

Table 4. **Ablations of agentic design and prompts** on CLEVR $_{100}$, a subset of 100 questions. We compare to single agent variant *No-API* which creates programs directly. We then ablate prompting by incrementally adding instructions to the agents used to define the API. The No-API Agent performs the worst and our prompting techniques add to VADAR's performance.

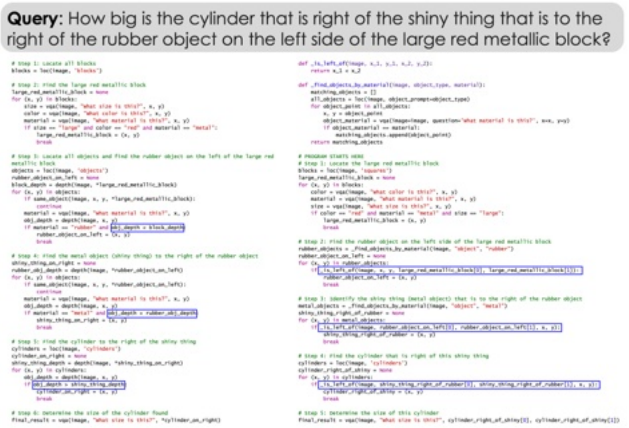
synthesis methods, we observe that VADAR and VisProg achieve comparable performance, while ViperGPT shows a drop in accuracy. A deeper dive into the output programs shows that VisProg relies on image-wide VQA calls in 34% of cases, whereas VADAR does so only 24% of the time. The limitations of GQA queries in evaluating 3D spatial reasoning highlight the need for our proposed benchmark, which better assesses 3D understanding and exposes the weaknesses of current methods.

4.3. Ablations

We turn to ablations to quantify the effectiveness of the agentic design and prompting in our approach. To reduce costs from GPT4o, we experiment on a randomly selected CLEVR subset. Tab. 4 compares the following variants:

No-API Agent is a single agent instructed to directly create programs for queries without defining an API of reusable methods. Comparison to this variant shows the value of an API. Fig. 5 shows a common reasoning error by the *No-API Agent*, which confuses depth with left/right; our approach, by implementing reusable methods, invokes the appropriately named method that is accurately implemented. The example reiterates that spatial reasoning relies on correctness, supporting VADAR's design to build an accurate API *before* program synthesis, over library learning, that discovers a potentially incorrect library *after* program synthesis.

API Agent is our approach without any prompting instructions or ICL examples. We incrementally add our two prompting techniques: (1) *Weak ICL* examples guide the Implementation Agent to use the pre-defined modules. (2) *Pseudo ICL* provides pseudo-code examples and instructions in *natural language* to the Implementation and Program Agent, respectively, that demonstrate how to handle intricate queries. We provide the prompts in the Appendix.



Query: How big is the cylinder that is right of the shiny thing that is to the right of the rubber object on the left side of the large red metallic block?

```python
# Step 1: Locate all blocks
blocks = loc(image, "blocks")

# Step 2: Find the large red metallic block
large_red_metallic_block = None
for (x, y) in blocks:
    size = vqa(image, "What size is this?", x, y)
    color = vqa(image, "What color is this?", x, y)
    material = vqa(image, "What material is this?", x, y)
    if size == "large" and color == "red" and material == "metal":
        large_red_metallic_block = (x, y)
        break

# Step 3: Locate all objects and find the rubber object on the left of the large red metallic block
objects = loc(image, "objects")
rubber_object_on_left = None
block_depth = depth(image, *large_red_metallic_block)
for (x, y) in objects:
    if same_object(image, x, y, *large_red_metallic_block):
        continue
    material = vqa(image, "What material is this?", x, y)
    obj_depth = depth(image, x, y)
    if material == "rubber" and obj_depth < block_depth
        rubber_object_on_left = (x, y)
        break

# Step 4: Find the metal object (shiny thing) to the right of the rubber object
shiny_thing_on_right = None
rubber_obj_depth = depth(image, *rubber_object_on_left)
for (x, y) in objects:
    if same_object(image, x, y, *rubber_object_on_left):
        continue
    material = vqa(image, "What material is this?", x, y)
    obj_depth = depth(image, x, y)
    if material == "metal" and obj_depth > rubber_obj_depth
        shiny_thing_on_right = (x, y)
        break

# Step 5: Find the cylinder to the right of the shiny thing
cylinders = loc(image, "cylinders")
cylinder_on_right = None
shiny_thing_depth = depth(image, *shiny_thing_on_right)
for (x, y) in cylinders:
    obj_depth = depth(image, x, y)
    if obj_depth > shiny_thing_depth
        cylinder_on_right = (x, y)
        break

# Step 6: Determine the size of the cylinder found
final_result = vqa(image, "What size is this?", *cylinder_on_right)
```

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

def _find_objects_by_material(image, object_type, material):
    matching_objects = []
    all_objects = loc(image, object_prompt=object_type)
    for object_point in all_objects:
        x, y = object_point
        object_material = vqa(image=image, question='What material is this?', x=x, y=y)
        if object_material == material:
            matching_objects.append(object_point)
    return matching_objects

# PROGRAM STARTS HERE
# Step 1: Locate the large red metallic block
blocks = loc(image, 'squares')
large_red_metallic_block = None
for (x, y) in blocks:
    color = vqa(image, "What color is this?", x, y)
    material = vqa(image, "What material is this?", x, y)
    size = vqa(image, "What size is this?", x, y)
    if color == "red" and material == "metal" and size == "large":
        large_red_metallic_block = (x, y)
        break

# Step 2: Find the rubber object on the left side of the large red metallic block
rubber_objects = _find_objects_by_material(image, "object", "rubber")
rubber_object_on_left = None
for (x, y) in rubber_objects:
    if _is_left_of(image, x, y, large_red_metallic_block[0], large_red_metallic_block[1]):
        rubber_object_on_left = (x, y)
        break

# Step 3: Identify the shiny thing (metal object) that is to the right of the rubber object
metal_objects = _find_objects_by_material(image, "object", "metal")
shiny_thing_right_of_rubber = None
for (x, y) in metal_objects:
    if _is_left_of(image, rubber_object_on_left[0], rubber_object_on_left[1], x, y):
        shiny_thing_right_of_rubber = (x, y)
        break

# Step 4: Find the cylinder that is right of this shiny thing
cylinders = loc(image, 'cylinders')
cylinder_right_of_shiny = None
for (x, y) in cylinders:
    if _is_left_of(image, shiny_thing_right_of_rubber[0], shiny_thing_right_of_rubber[1], x, y):
        cylinder_right_of_shiny = (x, y)
        break

# Step 5: Determine the size of this cylinder
final_result = vqa(image, "What size is this?", cylinder_right_of_shiny[0], cylinder_right_of_shiny[1])
```

(a) *No-API* Agent (b) VADAR

Figure 5. (a) The *No-API* agent produces longer programs and is prone to errors, often mistakenly using depth for left/right comparisons. (b) In contrast, our agentic VADAR creates shorter programs by leveraging methods from the API.

From Tab. 4 we observe that the No-API Agent performs the worst, while our prompting techniques via weak ICL examples and instructions achieve the best performance.

5. Limitations & Future Work

We introduce VADAR, an agentic approach that leverages LLM agents to dynamically create and expand a Pythonic API for complex 3D visual reasoning tasks. Our agents autonomously generate and implement functions, which are then utilized by the Program Agent to produce programs. This reuse of functions results in more accurate programs for complex queries. There is an extensive list of future directions to address current limitations of VADAR.

- VADAR often struggles with queries that require 5 or more inference steps, *e.g. "There is a yellow cylinder to the right of the cube that is behind the purple block; is there a brown object in front of it?"*. We provide the programs for these complex cases in the Appendix. Addressing such queries can be improved by leveraging advanced prompting strategies, an active research area that includes methods like CoT [41] and prompt chaining [42, 43].
- We show that VADAR attains high program accuracy (*e.g.*, 83.0% on CLEVR) but lower execution accuracy (53.6%) due to errors from the vision specialists. A potential enhancement would be to enable VADAR to dynamically choose its vision modules from a pool of available options based on empirical performance. Integrating the selection process with reinforcement learning or self-improvement mechanisms is a promising future direction.
- VADAR creates a program based solely on the input query, utilizing the image only during execution. Incorporating the image into the program synthesis process could improve accuracy, potentially improving performance on queries requiring five or more inference steps.

Acknowledgments

The project is funded by Meta through the LLM evaluation research grant and partly through Caltech's CAST program. We also thank Google's Gemma Academic program and OpenAI for granting us API credits for their LLMs.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. 1, 2, 5, 6, 8

[2] Anthropic. Claude, 2024. 1, 2, 5, 6

[3] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *ICCV*, 2015. 2, 3

[4] ARC-AGI. Arc prize, 2024. 3

[5] Garrick Brazil, Abhinav Kumar, Julian Straub, Nikhila Ravi, Justin Johnson, and Georgia Gkioxari. Omni3D: A large benchmark and model for 3D object detection in the wild. In *CVPR*, 2023. 2, 3, 5, 1

[6] Boyuan Chen, Zhuo Xu, Sean Kirmani, Brain Ichter, Dorsa Sadigh, Leonidas Guibas, and Fei Xia. Spatialvlm: Endowing vision-language models with spatial reasoning capabilities. In *CVPR*, 2024. 1, 2, 5, 6

[7] An-Chieh Cheng, Hongxu Yin, Yang Fu, Qiushan Guo, Ruihan Yang, Jan Kautz, Xiaolong Wang, and Sifei Liu. Spatialrgpt: Grounded spatial reasoning in vision-language models. In *NeurIPS*, 2024. 2

[8] Matt Deitke, Christopher Clark, Sangho Lee, Rohun Tripathi, Yue Yang, Jae Sung Park, Mohammadreza Salehi, Niklas Muennighoff, Kyle Lo, Luca Soldaini, et al. Molmo and pixmo: Open weights and open data for state-of-the-art multimodal models. *arXiv preprint arXiv:2409.17146*, 2024. 1, 3, 5, 6

[9] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. 2, 5, 6, 1

[10] Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lore Anaya Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: growing generalizable, interpretable knowledge with wake–sleep bayesian program learning. *Philosophical Transactions of the Royal Society A*, 2023. 3

[11] Arya Grayeli, Atharva Sehgal, Omar Costilla-Reyes, Miles Cranmer, and Swarat Chaudhuri. Symbolic regression with a learned concept library. In *NeurIPS*, 2024. 3

[12] Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In *CVPR*, 2023. 2, 5, 6, 8, 1

[13] Joy Hsu, Jiayuan Mao, and Jiajun Wu. Ns3d: Neuro-symbolic grounding of 3d objects and relations. In *CVPR*, 2023. 3

[14] Joy Hsu, Jiayuan Mao, Josh Tenenbaum, and Jiajun Wu. What's left? concept grounding with logic-enhanced foundation models. In *NeurIPS*, 2024. 3, 5, 6, 7

[15] Ziniu Hu, Ahmet Iscen, Aashi Jain, Thomas Kipf, Yisong Yue, David A Ross, Cordelia Schmid, and Alireza Fathi. Scenecraft: An llm agent for synthesizing 3d scenes as blender code. In *ICML*, 2024. 3

[16] Drew A Hudson and Christopher D Manning. Gqa: A new dataset for real-world visual reasoning and compositional question answering. In *CVPR*, 2019. 2, 3, 5, 7

[17] Dongfu Jiang, Xuan He, Huaye Zeng, Cong Wei, Max Ku, Qian Liu, and Wenhu Chen. Mantis: Interleaved multi-image instruction tuning, 2024. 5, 6, 1

[18] Justin Johnson, Bharath Hariharan, Laurens Van Der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *CVPR*, 2017. 2, 5

[19] Amita Kamath, Jack Hessel, and Kai-Wei Chang. What's" up" with vision-language models? investigating their struggle with spatial reasoning. *arXiv preprint arXiv:2310.19785*, 2023. 1, 2, 3

[20] Sahar Kazemzadeh, Vicente Ordonez, Mark Matten, and Tamara Berg. Referitgame: Referring to objects in photographs of natural scenes. In *EMNLP*, 2014. 3

[21] Wonjae Kim, Bokyung Son, and Ildoo Kim. Vilt: Vision-and-language transformer without convolution or region supervision. In *ICML*, 2021. 2

[22] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, et al. Segment anything. In *ICCV*, 2023. 3, 6

[23] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015. 3

[24] Jon M Laurent, Joseph D Janizek, Michael Ruzo, Michaela M Hinks, Michael J Hammerling, Siddharth Narayanan, Manvitha Ponnapati, Andrew D White, and Samuel G Rodriques. Lab-bench: Measuring capabilities of language models for biology research. *arXiv preprint arXiv:2407.10362*, 2024. 3

[25] Liunian Harold Li, Pengchuan Zhang, Haotian Zhang, Jianwei Yang, Chunyuan Li, Yiwu Zhong, Lijuan Wang, Lu Yuan, Lei Zhang, Jenq-Neng Hwang, et al. Grounded language-image pre-training. In *CVPR*, 2022. 2

[26] Yuan-Hong Liao, Rafid Mahmood, Sanja Fidler, and David Acuna. Reasoning paths with reference objects elicit quantitative spatial reasoning in large vision-language models. In *EMNLP*, 2024. 3

[27] Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Chunyuan Li, Jianwei Yang, Hang Su, Jun Zhu, et al. Grounding dino: Marrying dino with grounded pre-training for open-set object detection. *arXiv preprint arXiv:2303.05499*, 2023. 3

[28] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. The neuro-symbolic concept

learner: Interpreting scenes, words, and sentences from natural supervision. *ICLR*, 2019. 3

[29] Matthias Minderer, Alexey Gritsenko, Austin Stone, Maxim Neumann, Dirk Weissenborn, Alexey Dosovitskiy, Aravindh Mahendran, Anurag Arnab, Mostafa Dehghani, Zhuoran Shen, et al. Simple open-vocabulary object detection. In *ECCV*, 2022. 2

[30] Theo X Olausson, Alex Gu, Benjamin Lipkin, Cedegao E Zhang, Armando Solar-Lezama, Joshua B Tenenbaum, and Roger Levy. Linc: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. In *EMNLP*, 2023. 3

[31] Luigi Piccinelli, Yung-Hsu Yang, Christos Sakaridis, Mattia Segu, Siyuan Li, Luc Van Gool, and Fisher Yu. Unidepth: Universal monocular metric depth estimation. In *CVPR*, 2024. 3, 6

[32] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *ICML*, 2021. 2

[33] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. *IEEE TPAMI*, 2020. 2

[34] Jennifer J Sun, Megan Tjandrasuwita, Atharva Sehgal, Armando Solar-Lezama, Swarat Chaudhuri, Yisong Yue, and Omar Costilla Reyes. Neurosymbolic programming for science. In *NeurIPS 2022 Workshop on AI for Science: Progress and Promises*, 2022. 3

[35] Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. In *ICCV*, 2023. 2, 5, 6, 8, 1

[36] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023. 1, 2, 5, 6

[37] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving. In *CoLM*, 2024. 3

[38] Shengbang Tong, Ellis Brown, Penghao Wu, Sanghyun Woo, Manoj Middepogu, Sai Charitha Akula, Jihan Yang, Shusheng Yang, Adithya Iyer, Xichen Pan, et al. Cambrian-1: A fully open, vision-centric exploration of multimodal llms. *NeurIPS*, 2024. 1, 2, 3

[39] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. Houdini: Lifelong learning as program synthesis. *Advances in neural information processing systems*, 31, 2018. 3

[40] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *TMLR*, 2024. 3

[41] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*, 2022. 5, 8

[42] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. Promptchainer: Chaining large language model prompts through visual programming, 2022. 8

[43] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2022. Association for Computing Machinery. 8

[44] Jihan Yang, Shusheng Yang, Anjali W. Gupta, Rilyn Han, Li Fei-Fei, and Saining Xie. Thinking in Space: How Multimodal Large Language Models See, Remember and Recall Spaces. *arXiv preprint arXiv:2412.14171*, 2024. 3, 5, 1, 2

[45] Yan Zeng, Xinsong Zhang, and Hang Li. Multi-grained vision language pre-training: Aligning texts with visual concepts. *arXiv preprint arXiv:2111.08276*, 2021. 2