PARSE BEFORE YOU EXECUTE: EFFICIENT SPATIAL REASONING WITH EXECUTABLE PRIMITIVES

Anonymous authors
Paper under double-blind review

ABSTRACT

Spatial reasoning in Vision-Language Models (VLMs) has increasingly benefited from tool-integrated pipelines that ground computation in explicit geometry. However, existing systems often compile queries into relatively open-ended artifacts, such as code, task constraints, tool plans, or gathered evidence, leaving what must be resolved before execution under-specified. This can lead to unnecessary tool acquisition and iterative refinement, while still producing plausible reasoning traces from incomplete spatial commitments. We identify this failure mode as under-committed query compilation. To address it, we propose **PEPS**, a training-free agentic paradigm **P**arsing **E**xecutable **P**rimitives for **S**patial reasoning under a typed FESM schema over Frame, Entity, State, and Metric. PEPS follows a *parse-before-execute* principle: the Parser commits each query to a compact set of executable requirements; the Executor selectively grounds the required primitives through fixed geometric APIs and deterministic computation; and the Verifier checks execution sufficiency, enabling early termination or targeted repair. Extensive experiments across challenging spatial reasoning benchmarks show that PEPS consistently improves reasoning accuracy while substantially reducing refinement rounds, API cost, and end-to-end latency over strong tool-integrated baselines. These results demonstrate that explicit requirement commitment can make tool-integrated spatial reasoning not only more reliable, but also more execution-efficient. Source code is available at `https://anonymous.4open.science/r/PEPS-A31A`.

1 INTRODUCTION

Spatial reasoning is rarely a matter of exhaustively describing everything in a scene. Cognitive studies suggest that humans reason about space in a question-driven way, selecting the scene structure that is relevant to the task at hand (Tversky, 1991; Byrne & Johnson-Laird, 1989). Someone moving a sofa through a doorway, for instance, needs the doorway width, the sofa's largest dimension, and whether it can be tilted, while many other visual details are irrelevant. This observation suggests a useful design principle for spatial reasoning systems: *before perception tools are invoked or geometric computation begins, a spatial reasoning system should first commit to what the query actually requires it to know.*

Recent spatial reasoning systems have made substantial progress by moving part of the reasoning process out of unconstrained VLM inference and into explicit tools Surís et al. (2023); Gupta & Kembhavi (2023); Jia et al. (2025). Prior work has shown the value of offloading geometric computation to deterministic modules Luo et al. (2026); Chen et al. (2025), expanding or adapting tool use through APIs Marsili et al. (2025); Wu et al. (2025b), and curating sufficient evidence for downstream reasoning Guo et al. (2025). Together, these advances suggest that reliable spatial reasoning depends not only on stronger models, but also on how perception, computation, and evidence acquisition are organized. Yet existing systems determine what is needed through open-ended plans, programs, formalizations, or iterative evidence requests. When the required information is not committed before execution, the agent may acquire unnecessary evidence, invoke redundant tools, or repeatedly refine an initially incomplete reasoning trajectory.

The missing piece is therefore not only another tool, but an explicit interface between the query and its execution. In existing spatial reasoning systems, the artifact compiled from a query, whether free-form code, a natural-language formalization, a tool-selection plan, or an information set, often remains in an open output space Marsili et al. (2025); Luo et al. (2026); Chen et al. (2025). Such artifacts may specify how to proceed without explicitly committing to what must be resolved before

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

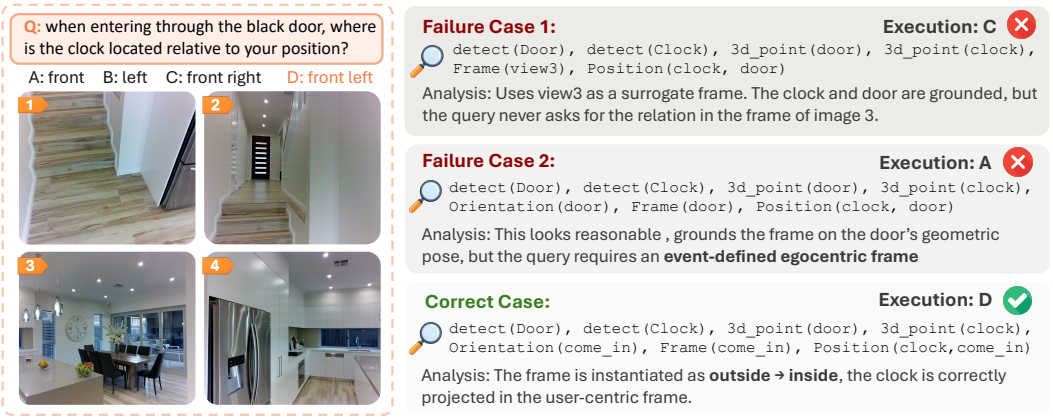


Figure 1: Plausible but incomplete requirement commitments can mislead spatial reasoning.

execution. As a result, a system can produce a plausible execution trace even when the required spatial commitments are wrong or incomplete. Figure 1 illustrates this failure mode: the tools are sufficient and the geometry is correct, yet execution fails because the query is compiled into the wrong requirements. We call this *under-committed query compilation*. Beyond correctness, such incomplete commitments also create computational redundancy: missing requirements must be discovered downstream through additional acquisition, replanning, or refinement.

We therefore follow a simple principle: *parse before you execute*. Rather than asking a VLM to freely generate what to do next, we first compile the query into a compact set of executable spatial primitives retrieved from a predefined library. Each primitive specifies one requirement that must be resolved before deterministic computation, such as which reference frame should be used, which entities must be grounded, which geometric states must be acquired, and which measurable relation should be computed. We organize these requirements into a four-part schema, **FESM**: Frame, Entity, State, and Metric. This typed interface makes the execution boundary explicit: only required primitives need to be instantiated, while unresolved requirements can be identified and repaired without restarting an open-ended reasoning process. Parsing therefore serves not only as an inspectable commitment, but also as a mechanism for selective execution and early termination.

We instantiate this principle in **PEPS**, a multi-agent framework for **P**arsing **E**xecutable **P**rimitives for **S**patial Reasoning. The Parser commits each query to a compact FESM requirement set. The Executor selectively grounds the required primitives through geometric APIs and deterministic computation, reusing available results whenever possible. The Verifier checks whether the resulting answer is fully supported by the query, visual evidence, and execution trace; sufficiently supported traces terminate immediately, while rejected traces receive typed feedback that distinguishes missing-slot failures from genuine primitive gaps. This compile-execute-verify design targets both reasoning reliability and execution efficiency by avoiding unnecessary acquisition and repeated refinement. We evaluate PEPS on MMSI-Bench Yang et al. (2025c), ViewSpatial-Bench Li et al. (2025), and OMNI3D-Bench Marsili et al. (2025). PEPS consistently improves spatial reasoning performance across different VLM backbones and, compared with the strongest iterative tool-integrated baseline, reduces API cost by 15.0-23.5% and mean end-to-end latency by 32.3-38.2%, with 83.0-93.0% of queries terminating after a single round. Our contributions are threefold:

(1) We characterize under-committed query compilation as a failure mode in tool-integrated spatial reasoning, where plausible intermediate artifacts may leave query-required spatial commitments unresolved. This motivates making query-to-requirement commitment explicit before execution.

(2) We introduce PEPS, a *parse-before-execute* framework, and FESM, a typed interface over Frame, Entity, State, and Metric. Compact executable primitives enable selective tool execution, deterministic computation, typed failure repair, and early termination rather than open-ended iterative acquisition.

(3) Extensive experiments show that explicit requirement commitment yields consistent gains across benchmarks and VLM backbones. Beyond accuracy improvements, PEPS reduces API cost, refinement rounds, and end-to-end latency over iterative tool-integrated baselines. Further analyses show that primitive parsing also benefits existing agentic pipelines as a committed-compilation front end.

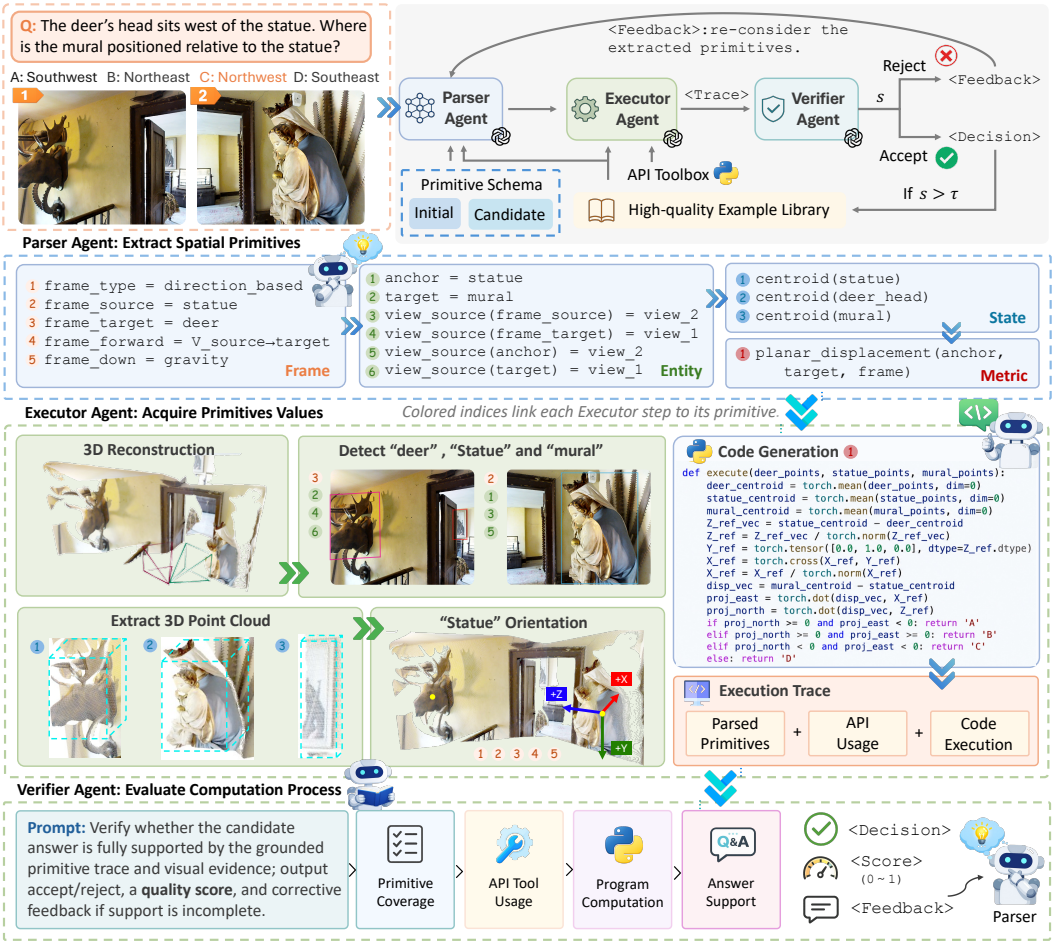


Figure 2: **Overall Paradigm.** The Parser retrieves Frame, Entity, State, and Metric primitives from the query. The Executor grounds them with a fixed geometric toolbox and deterministic code, producing a candidate answer and trace. The Verifier outputs `<Decision>`, `<Score>`, and `<Feedback>`; rejected candidates are refined, and high-scoring accepted ones are stored for reference.

2 METHODOLOGY

Problem Formalization. We consider language-conditioned spatial reasoning over a scene observed from one or multiple views. Let $\mathcal{I} = \{I_1, \ldots, I_M\}$ denote the input images, q a natural-language spatial query, and a^* the ground-truth answer. Given a typed primitive library $\mathcal{L}$ over Frame, Entity, State, and Metric, our goal is to recover a compact sufficient executable primitive set $\mathcal{P} \subseteq \mathcal{L}$ that specifies the requirements needed to answer q through deterministic geometric computation.

PEPS realizes this objective through a closed loop of parsing, execution, and verification. The Parser retrieves a candidate requirement set $\mathcal{P} \subseteq \mathcal{L}$, the Executor selectively instantiates the unresolved primitives through a fixed API toolbox $\mathcal{S}$, and the Verifier evaluates whether the resulting candidate is sufficiently supported by the query, visual evidence, and execution trace:

$$\mathcal{P} = \mathcal{A}_{\text{parse}}(\mathcal{I}, q, \mathcal{L}), \qquad (\hat{a}, T) = \mathcal{A}_{\text{exec}}(\mathcal{I}, q, \mathcal{P}, \mathcal{S}), \qquad (v, s, F) = \mathcal{A}_{\text{verify}}(\mathcal{I}, q, \hat{a}, T), \quad (1)$$

where T is the execution trace, $v \in \{\texttt{accept}, \texttt{reject}\}$ is the verification decision, $s \in \mathbb{R}$ is the quality score, and F is the structured feedback returned when $v = \texttt{reject}$. The loop terminates once a candidate is accepted as the final answer or reaches the maximum loop iteration limitation.

2.1 SPATIAL PRIMITIVES SCHEMA CONSTRUCTION

What must be explicitly resolved before a spatial query can be executed deterministically? To answer this, we construct a typed FESM schema as an executable interface between query in-

terpretation and tool execution (Refer Appendix G). FESM is not a taxonomy of surface spatial relations, but follows their execution dependencies as illustrated in Figure 3. Frame primitives determine the reference coordinate system, Entity primitives identify what to ground, State primitives specify the geometric variables to acquire, and Metric primitives define the relation to compute. Together, they specify what must be instantiated while avoiding unnecessary acquisition. We induce the initial FESM schema with a two-stage prompting pipeline over questions sampled from MMSI-Bench Yang et al. (2025c), ViewSpatial-Bench Li et al. (2025), and OMNI3D-Bench Marsili et al. (2025). At the per-question level, GPT-4o OpenAI (2024) decomposes each query into candidate FESM requirements without solving the question or proposing tool calls. These decompositions are then aggregated by merging equivalent requirements, parameterizing object-specific variables, removing candidates unsupported by the tool suite, and recording supporting question IDs. Detailed prompts are provided in Appendix Tables 14 and 15. We further evaluate schema transfer to MindCube-Tiny Yin et al. (2025) and OmniSpatial Jia et al. (2025) in Section 3.4.

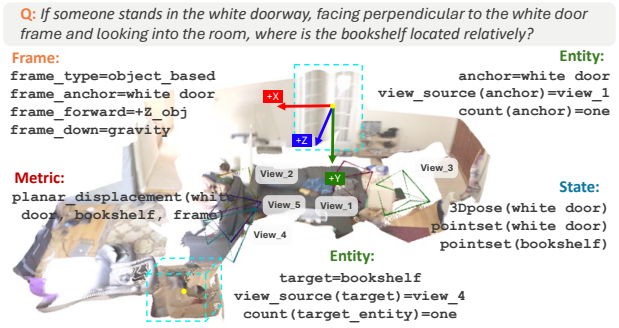


Figure 3: FESM-dimensional decomposition of a query.

2.2 Parser Agent

The Parser Agent is responsible for committed compilation. Given the input images $\mathbf{I}$, query q, and current library $\mathcal{L}$, it retrieves a structured requirement set $\mathcal{P} = \mathcal{A}_{\text{parse}}(\mathbf{I}, q, \mathcal{L})$ that specifies what must be grounded before deterministic spatial computation. Its output is restricted to primitive instances in $\mathcal{L}$, making the parsed result an explicit and inspectable statement of the query's representational requirements. This stage is separated from execution because requirement identification should not be implicitly absorbed into downstream planning. Otherwise, the system may substitute tool-available evidence for query-required evidence, yielding plausible but incomplete formalizations. The Parser avoids this by committing the query to a bounded requirement set before any tool call or computation.

The retrieval follows the dependency order of spatial reasoning: Frame, Entity, State, then Metric. The Parser first resolves the reference frame, then identifies the relevant entities and grounding constraints, next selects the geometric states needed for the queried relation under a coarse-first policy, and finally specifies the metric that directly determines the answer. This makes $\mathcal{P}$ a dependency-consistent requirement set rather than an open-ended factor list. The Parser does not choose tools, instantiate values, or perform computation, which keeps its output independently auditable from execution success. Detailed prompt for Parser is provided in Appendix Table 16.

2.3 Executor Agent

The Executor Agent operationalizes the primitives retrieved by the Parser by invoking a curated suite of geometric APIs, as summarized in Table 1, bridging the typed FESM requirements with measurable 3D geometry. See Table 17 for the role prompt and Appendix A for module implementation details.

3D Scene Reconstruction. This module bridges the gap between sparse 2D observations and a unified 3D geometric substrate. Here in our experiments, we adopt VGGT (Wang et al., 2025a) to estimate camera parameters, depth maps, and generate a unified 3D point cloud. This reconstruction serves as the foundational workspace upon which all subsequent primitive values are grounded. For example, coarse entity positions, orientations, and anchor points are all extracted relative to this shared scene representation, which provides a consistent basis for subsequent deterministic computation.

Reference Frame Grounding. The primary challenge of spatial reasoning is not only what relation to compute, but under which reference frame that relation should be interpreted Chen et al. (2025); Guo et al. (2025). We treat this as a reference-frame grounding problem and formalize a reference frame as a 3D coordinate system $\mathcal{R}$ with origin $O_{\mathcal{R}}$ and orthogonal basis vectors $(\mathbf{x}_{\mathcal{R}}, \mathbf{y}_{\mathcal{R}}, \mathbf{z}_{\mathcal{R}})$. In PEPS, the retrieved Frame primitives explicitly determine this coordinate system: `frame_type` selects a

Table 1: Overview of Tool APIs, underlying visual foundation models, and their core skills.

API Tools	Visual Models	Skills
`reconstruct`	VGGT Wang et al. (2025a)	3D reconstruction; camera parameters; world points; object alignment
`detect`	GroundingDINO Liu et al. (2024)	Object detection; bounding boxes; labels
`project_box_to_3d_points`	SAM-2 Ravi et al. (2024)	2D→3D projection; pixel mask filtering; point extraction
`predict_obj_pose`	Orient Anything Wang et al. (2024)	6-DoF pose estimation; object-to-world transformation; frame setup
`estimate_scale`	MoGe-2 Wang et al. (2025c) + VGGT	Metric scale computation; align depth maps; real-world units
`ocr`	EasyOCR Jaided (2020)	Optical character recognition; bounding boxes
`analyze_motion`	Farneback optical flow Farnebäck (2003)	Pixel-level motion; camera movement inference
`code`	Python sandbox	Python execution

camera_based, *object_based*, or *direction_based* frame, while `frame_view`, `frame_anchor`, or `frame_source`/`frame_target` specifies its grounding. This converts viewpoint and directional ambiguities into checkable and executable commitments. See Table 10 for details.

Deterministic Program Generation. Once the required primitives are instantiated, the Executor invokes the `code` API for deterministic geometric computation using only $\mathcal{P}$ and the filled workspace, with geometric knowledge from GCA Chen et al. (2025). It cannot request additional perception, introduce undeclared variables, or revise the committed requirements, preventing fallback to open-ended VLM inference as the final stage. The generation prompt is provided in Table 18.

2.4 Verifier Agent

The Verifier Agent closes the loop between committed compilation and execution by assessing the candidate $(\hat{a}, \mathcal{T})$ and producing (v, s, F). Rather than solving the query again, it checks whether the answer is fully supported by the query, visual evidence, and primitive-grounded execution trace. If the required primitives are sufficiently instantiated and the computation is supported, $v = \text{accept}$ and execution terminates without further refinement. High-quality accepted traces with $s > \tau$ are stored in the Example Library for future retrieval. Otherwise, the Verifier returns typed feedback that enables targeted repair rather than restarting open-ended reasoning. We distinguish two cases:

Missing-slot feedback indicates that the schema is expressive enough, but some required primitives are missing or insufficiently instantiated, e.g., incomplete frame grounding or metric prerequisites. The Parser then refines only the unresolved requirements using primitives already in the library.

Primitive-gap feedback indicates that the current schema cannot express a query requirement. The Parser may propose temporary primitive hypotheses, which enter the candidate pool and are promoted only after repeated reuse and verified downstream improvement. In this way, the Verifier provides both query-level corrective supervision and schema-level signals for controlled library expansion.

3 Experiments

3.1 Experimental Setup

Benchmarks. We evaluate PEPS on three complementary benchmarks. **MMSI-Bench** (Yang et al., 2025c) is a manually curated multi-image benchmark for real-world spatial reasoning, covering camera pose, object layout, region relations, and their compositions. Its multi-step subset tests consistent spatial reasoning over longer chains. **ViewSpatial-Bench** (Li et al., 2025) focuses on viewpoint-sensitive spatial understanding, particularly egocentric versus allocentric reasoning, across multiple task categories and changing views. We further evaluate on **OMNI3D-Bench** Marsili et al. (2025), a single-view benchmark for fine-grained 3D spatial reasoning from monocular observations.

Baselines. We compare PEPS against four groups of baselines: (1) *Proprietary LLMs*, including GPT-4o OpenAI (2024), o3 OpenAI (2025), and the Gemini-2.5 family Google DeepMind (2025); (2) *Open-source VLMs*, including Llama-3.2-Vision Meta (2024), LLaVA-OneVision Li et al. (2024), Qwen2.5 Bai et al. (2025)/Qwen3-VL Yang et al. (2025a), and InternVL2.5 Chen et al. (2024)/InternVL3 Zhu et al. (2025); (3) *Training-based 3D-VLMs*, including Video-3D-LLM Zheng et al. (2025), LLaVA-3D Zhu et al. (2024), VLM-3R Fan et al. (2025), and VILASR Wu et al. (2025a), which incorporate explicit 3D representations or 3D-enhanced inference; and (4) *Tool-integrated agents*, including VADAR Marsili et al. (2025), pySpatial Luo et al. (2026), GCA Chen et al. (2025), MSSR Guo et al. (2025), ViperGPT Surís et al. (2023), and VisProg Gupta & Kembhavi (2023).

Table 2: **Comparison with baselines on MMSI-Bench and ViewSpatial-Bench.** The best results are shown in **bold**, and the second-best are underlined. Results for PEPS and MSSR are reported as Mean$_{\pm \text{Std Dev}}$ over 3 independent runs; other entries are single-run or reported baseline results.

Method	MMSI-Bench					ViewSpatial-Bench		
	Positional Relationship	Multi-Step Reasoning	Attribute	Motion	Overall	Camera Based	Person Based	Overall
Random Guessing	25.0	25.0	25.0	25.0	25.0	25.5	27.1	26.3
Proprietary LLMs								
o3 OpenAI (2025)	45.8	34.9	41.6	32.0	41.0	51.3	51.0	**51.1**
Gemini 2.5 Pro Google DeepMind (2025)	39.0	34.3	36.1	33.3	36.9	44.3	41.6	43.0
Gemini 2.5 Flash Google DeepMind (2025)	37.4	30.3	34.5	33.4	35.0	41.6	36.9	38.4
GPT-4o OpenAI (2024)	28.0	30.8	32.3	36.0	30.3	33.6	36.3	35.0
Open-source VLMs								
Llama-3.2-11B-Vision Meta (2024)	26.9	19.2	20.0	27.3	24.5	23.7	33.6	28.8
LLaVA-OneVision-7B Li et al. (2024)	28.0	11.6	27.7	26.7	24.5	28.5	26.6	27.5
Qwen2.5-VL-3B Bai et al. (2025)	29.5	23.2	23.8	22.7	26.5	39.8	32.1	35.8
Qwen2.5-VL-7B Bai et al. (2025)	25.9	25.8	21.5	30.0	25.9	40.6	33.4	36.9
Qwen2.5-VL-72B Bai et al. (2025)	31.2	27.3	36.2	28.7	30.7	47.6	38.9	43.1
Qwen3-VL-4B Yang et al. (2025a)	32.8	27.8	31.0	29.7	31.1	42.3	35.9	39.0
Qwen3-VL-8B Yang et al. (2025a)	32.6	30.3	27.8	29.9	31.1	45.1	37.5	41.2
InternVL2.5-8B Chen et al. (2024)	29.9	30.3	27.0	24.0	28.7	46.5	40.2	43.2
InternVL2.5-2B Chen et al. (2024)	30.5	26.8	29.3	26.7	29.0	32.8	37.0	35.0
InternVL3-14B Zhu et al. (2025)	25.5	29.3	26.9	28.0	39.6	47.1	33.9	40.3
Trained-based 3D-VLMs								
Video-3D-LLM Zheng et al. (2025)	25.5	25.8	22.4	25.2	25.3	24.4	35.8	30.3
LLaVA-3D Zhu et al. (2024)	29.5	19.7	32.0	30.1	28.0	39.6	28.3	33.8
VLM-3R Fan et al. (2025)	33.0	30.3	31.2	31.6	32.0	43.8	34.9	39.2
VILASR Wu et al. (2025a)	35.9	23.2	26.0	21.0	29.8	42.4	34.4	38.2
Tool Integrated Agent								
VADAR Marsili et al. (2025)	32.8	22.7	24.3	27.7	28.9	34.2	33.2	33.7
pySpatial Luo et al. (2026)	29.9	31.4	38.1	37.0	32.3	37.5	40.6	39.1
GCA Chen et al. (2025)	36.7	32.5	36.4	38.5	36.1	45.2	49.4	47.4
MSSR Guo et al. (2025)	49.4 $\pm$0.2	47.8 $\pm$0.2	47.1 $\pm$0.3	44.9 $\pm$0.2	48.1 $\pm$0.2	51.0 $\pm$0.1	**54.4** $\pm$0.2	51.8 $\pm$0.4
PEPS (Ours)	**51.3** $\pm$0.2	**48.2** $\pm$0.3	**52.6** $\pm$0.4	**52.9** $\pm$0.4	**51.1** $\pm$0.2	**51.9** $\pm$0.3	52.9 $\pm$0.1	**52.4** $\pm$0.2

Implementation Details. We implement PEPS as a modular multi-agent pipeline, where all three roles are instantiated with GPT-4o through the commercial API, using role-specific prompts and structured JSON interfaces. We set the maximum loop budget to 3 and Example Library threshold $\tau = 0.7$. Perceptual tools and geometric computation modules are executed locally. All experiments are conducted on 2 NVIDIA A100 GPUs. We reproduce baseline results using their officially released code and evaluation settings. Full hyperparameter choices are provided in Appendix A.4.

3.2 Main Result

PEPS generalizes strongly across benchmark settings. On MMSI-Bench, PEPS reaches **51.1%** overall accuracy, exceeding the strongest proprietary baseline o3 at 41.0% by 10.1 points and outperforming recent tool-integrated methods such as GCA at 36.1% and MSSR at 48.1%. This advantage remains stable on ViewSpatial-Bench, where PEPS achieves a new best overall score of **52.4%**, surpassing GCA at 47.4% and MSSR at 51.8%. The same pattern carries over to OMNI3D-Bench, where PEPS obtains the best overall result of **49.0%**. Taken together, these results show that the gain of PEPS is not tied to a single benchmark format, but transfers across multi-image, viewpoint-sensitive, and monocular 3D spatial reasoning settings.

Table 3: **Performance comparison on OMNI3D-Bench**.

Method	Counting	Float-MRA	Yes/No	Multiple Choice	Overall
Generic VLMs					
GPT-4o	28.1	35.5	66.7	57.2	42.9
Claude3.5-Sonnet	22.4	20.6	62.2	50.6	32.2
Llama-3.2	24.3	19.3	47.5	27.4	25.6
Gemini1.5-Pro	25.2	28.1	46.2	37.6	32.0
Gemini1.5-Flash	24.3	27.6	51.1	52.9	35.0
Qwen2-VL-7B-Instruct	12.9	21.5	58.7	33.7	28.0
LLaVA-OV-7B-Chat	22.9	26.8	60.0	27.9	31.4
SpaceMantis	20.0	21.7	50.6	48.2	30.3
Visual Programming Approaches					
ViperGPT	20.0	15.4	56.0	42.4	26.7
VisProg	2.9	0.9	54.7	25.9	13.5
VADAR	21.7	35.5	56.0	57.6	40.4
pySpatial	22.9	38.6	72.0	54.7	44.2
GCA	25.6	40.2	70.1	59.3	45.9
MSSR	28.4 $\pm$0.1	42.7 $\pm$0.3	**73.8** $\pm$0.5	59.1 $\pm$0.4	48.2 $\pm$0.4
PEPS (Ours)	**28.8** $\pm$0.4	**44.0** $\pm$0.7	73.5 $\pm$0.2	**59.5** $\pm$0.4	**49.0** $\pm$0.5

PEPS is especially effective on viewpoint-sensitive and computation-heavy reasoning. On MMSI-Bench, PEPS performs best on Positional Relationship, Multi-Step Reasoning, and Motion, reaching 51.3%, 52.6%, and 52.9%, respectively, suggesting that executable primitive parsing benefits both basic localization and compositional reasoning. On ViewSpatial-Bench, it performs consistently across Camera Based and Person Based settings (51.9% and 52.9%), maintaining stable spatial commitment across ex-

Table 4: **Average Per-query API Usage and Cost on MMSI-Bench.**

Method	API Cost (USD/query)	API Calls (calls/query)	Non-cached Input (tokens/query)	Cached Input (tokens/query)	Total Input (tokens/query)	Output (tokens/query)
MSSR (GPT-4o)	0.1800	4.30	7,200.00	1,600.00	8,800.00	16,000.00
PEPS (GPT-4o)	0.1530 ↓15.0%	3.60 ↓16.3%	6,000.00 ↓16.7%	4,800.00 ↑200.0%	10,800.00 ↑22.7%	13,200.00 ↓17.5%
GPT-5	0.0862	1.00	4,900.00	900.00	5,800.00	8,000.00
MSSR (GPT-5)	0.1562	3.90	6,400.00	1,800.00	8,200.00	14,800.00
PEPS (GPT-5)	0.1231 ↓21.2%	3.30 ↓15.4%	5,200.00 ↓18.8%	4,800.00 ↑166.7%	10,000.00 ↑22.0%	11,600.00 ↓21.6%
Gemini 3.6 Flash	0.0557	1.00	4,500.00	1,000.00	5,500.00	6,500.00
MSSR (Gemini 3.6 Flash)	0.1005	3.64	5,800.00	2,200.00	8,000.00	12,200.00
PEPS (Gemini 3.6 Flash)	0.0769 ↓23.5%	3.24 ↓11.0%	4,800.00 ↓17.2%	4,700.00 ↑113.6%	9,500.00 ↑18.8%	9,200.00 ↓24.6%

Table 5: **Average Per-query Computational Overhead on MMSI-Bench.**

Method	GPU Memory (GB/process)	Mean E2E Latency (s/query)	Median E2E Latency (s/query)	P95 E2E Latency textbf(s/query)	Average Iterations (rounds/query)	1-Iteration (%)	2-Iteration (%)	3-Iteration (%)
MSSR (GPT-4o)	7.20	86.75	79.60	137.40	2.15	15.0%	55.0%	30.0%
PEPS (GPT-4o)	9.80 ↑36.1%	53.60 ↓38.2%	46.20 ↓42.0%	92.80 ↓32.5%	1.20 ↓44.2%	83.0% ↑68.0%	14.0% ↓41.0%	3.0% ↓27.0%
GPT-5	0.00†	34.80	31.20	60.50	1.00	100.0%	0.0%	0.0%
MSSR (GPT-5)	7.20	98.75	96.40	165.20	1.95	25.0%	55.0%	20.0%
PEPS (GPT-5)	9.80 ↑36.1%	66.83 ↓32.3%	61.50 ↓36.2%	117.80 ↓28.7%	1.10 ↓43.6%	91.5% ↑66.5%	7.0% ↓48.0%	1.5% ↓18.5%
Gemini 3.6 Flash	0.00	23.40	20.80	42.60	1.00	100.0%	0.0%	0.0%
MSSR (Gemini 3.6 Flash)	7.20	72.40	67.90	125.60	1.82	30.0%	58.0%	12.0%
PEPS (Gemini 3.6 Flash)	9.80 ↑36.1%	45.70 ↓36.9%	40.80 ↓39.9%	82.40 ↓34.4%	1.08 ↓40.7%	93.0% ↑63.0%	6.0% ↓52.0%	1.0% ↓11.0%

ternally anchored and human-centered frames. On OMNI3D-Bench, PEPS achieves 44.0% on Float-MRA, outperforming GCA (40.2%) and MSSR (42.7%) on precise metric computation.

3.3 Parse Before Execution Reduces Reasoning Overhead

PEPS achieves higher accuracy with substantially less iterative reasoning overhead. As shown in Tables 4 and 5, PEPS reduces API cost by 15.0-23.5% and mean end-to-end latency by 32.3-38.2% over MSSR across GPT-4o, GPT-5, and Gemini 3.6 Flash. This efficiency gain follows from the *parse-before-execute* design: by committing to executable requirements before tool invocation, 83.0-93.0% of queries terminate after one round, reducing average refinement from 1.82-2.15 rounds for MSSR to 1.08-1.20 for PEPS. Although PEPS uses more cached context and GPU memory, fewer API calls and output tokens yield lower practical inference cost and latency. Together with the consistent accuracy gains, these results show that explicit requirement commitment enables a stronger accuracy-efficiency trade-off rather than relying on repeated downstream refinement.

3.4 Ablation Study

Effect of Spatial Primitives. We examine whether the gains come from the FESM primitive interface itself. Table 6 compares PEPS with three weakened variants: Free-form uses natural-language descriptions; Flat-schema removes the Frame, Entity, State, and Metric hierarchy; and Fixed-schema disables Verifier-driven schema updates. All three variants perform worse than the full model. The largest drop comes from free-form parsing, showing that open-ended requirement descriptions are less reliable than executable primitive commitments. The flat-schema drop shows that the FESM hierarchy provides useful dependency structure, while the fixed-schema drop suggests that controlled schema expansion improves coverage. We also prepend the same primitive parsing stage to baselines while keeping their downstream pipelines unchanged; all baselines improve as shown in Figure 4(a). This indicates that FESM parsing acts as a general committed-compilation front end.

Component Contribution. Table 7 illustrates the contribution of each core component by progressively weakening the full system along four axes: multi-agent role decomposition, verification, corrective feedback, and example-based experience retrieval. The largest weakness appears when verification is removed, indicating that PEPS relies on trace-level inspection to prevent incomplete primitive commitments from passing into the final answer. Corrective feedback is also critical because it turns verification from passive filtering into active repair, allowing the Parser to recover missing

Table 6: Ablation study on spatial primitives. "Flat-schema" removes the FESM hierarchy, and "Fixed-schema" disables the primitive-pool update mechanism.

Method	MMSI	OMNI3D
GPT-4o	30.3	42.9
Ours (Full)	**51.1**	**49.0**
Free-form	42.4	44.8
Flat-schema	49.2	47.7
Fixed-schema	50.7	48.5

Table 7: Ablation study on the contribution of each component. "Multi-Agent" denotes explicit role decomposition into three sub-tasks, "Feedback" denotes corrective feedback returned by the Verifier, and "Example" denotes retrieval from the Example Library of high-quality historical candidates.

Multi-Agent	Verifier	Feedback	Example	MMSI-Bench
✗	✓	✓	✓	49.2
✓	✗	✗	✗	47.6
✓	✓	✗	✗	48.9
✓	✓	✓	✗	50.4
✓	✓	✓	✓	**51.1**

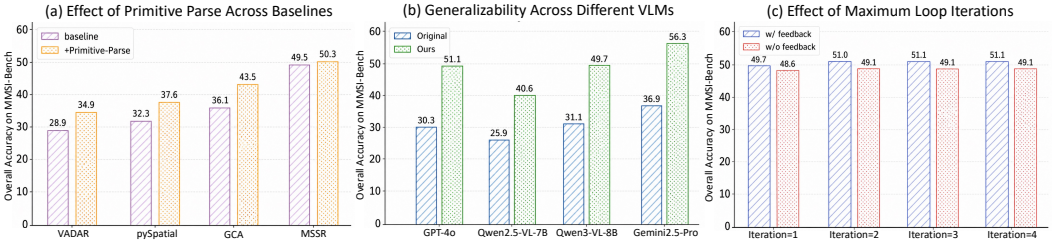


Figure 4: **Ablation studies.** We evaluate three scenarios: (a) prepending primitive parsing to existing spatial reasoning agentic baselines; and (b) applying PEPS to different VLM backbones; and (c) varying the maximum loop budget, with or without corrective Verifier feedback.

slots instead of merely retrying. Role separation improves reliability by keeping requirement parsing, value acquisition, and answer checking from collapsing into one loosely constrained reasoning step.

Cross-Dataset Generalizability of Primitive Schema. We test whether the primitive schema transfers to unseen benchmarks. As shown in Table 8, the fixed source-induced schema consistently outperforms free-form parsing on MindCube-Tiny Yin et al. (2025) and OmniSpatial Jia et al. (2025). This suggests that FESM captures reusable execution requirements rather than benchmark-specific cues, remaining effective as scene composition and query style shift across datasets. Importantly, this transfer requires no benchmark-specific schema reconstruction, supporting FESM as a reusable interface.

Table 8: Cross-Dataset Generalizability of Primitive Schema.

Method	MindCube-Tiny	OmniSpatial
GPT-4o	35.8	51.5
Free-form	61.3	62.1
PEPS	73.7	69.8

Sensitivity Analysis on 3D Reconstruction Backbones. We evaluate whether PEPS depends on a specific 3D reconstruction backbone by replacing VGGT with π^3 Wang et al. (2025d) and CUT3R Wang et al. (2025b), with others unchanged. As shown in Table 9, all reconstruction backbones substantially outperform the GPT-4o baseline, indicating that the main gain does not depend on a particular reconstruction model. π^3 gives the best result, while VGGT and CUT3R remain close. This suggests that PEPS is robust to the geometric substrate as long as the reconstruction provides usable camera poses and object-level 3D states.

Table 9: Sensitivity analysis of different 3D reconstruction backbones.

Method	MMSI-Bench
GPT-4o	30.3
PEPS w/ VGGT	51.1
PEPS w/ π^3	51.8
PEPS w/ CUT3R	50.5

Generalizability Across VLM Backbones. Figure 4(b) suggests that PEPS functions as model-agnostic rather than a prompt tuned to one particular backbone. Across different VLMs, the primitive interface consistently provides a more stable reasoning scaffold by narrowing each model's role from open-ended spatial inference to structured requirement retrieval, grounding, and verification. This is especially important for weaker backbones, where unconstrained planning is more likely to drift, but it also benefits stronger models by reducing ambiguity in reference frames and metric selection.

Effect of Iteration Budget. We study the effect of the maximum loop budget by varying the number of refinement rounds with and without corrective feedback. As displayed in Figure 4(c), performance improves mainly in the early rounds and quickly saturates. The gap between the feedback and no-feedback setting shows that the loop is not simply benefiting from repeated attempts, but from targeted repair of incomplete primitive commitments. And the limited gain from later rounds suggests

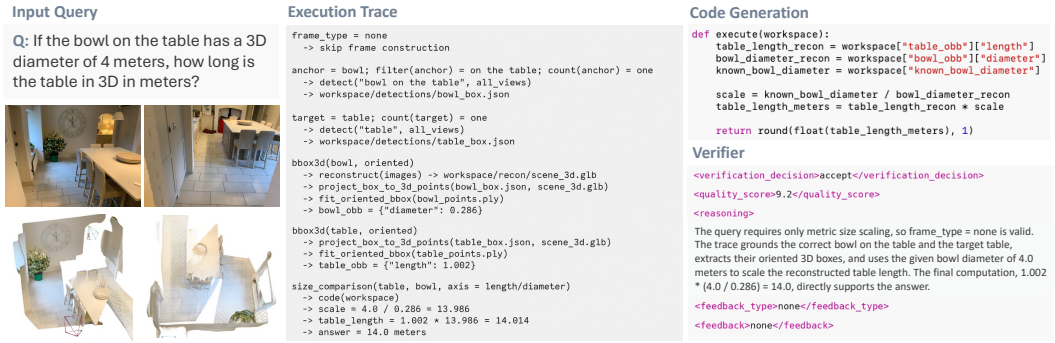


Figure 5: **Compact Qualitative Example.** More detailed examples are provided in Appendix F.

that remaining failures are less likely to be fixed by further parsing refinement and more likely to stem from perception errors, reconstruction noise, or genuine schema gaps. This indicates that PEPS functions as a fast diagnostic repair loop rather than an open-ended self-refinement process.

3.5 Failure Cases Analysis

We manually analyze about 100 failed examples and attribute each to its earliest dominant error source, as shown in Figure 6. Notably, most failures arise before final deterministic computation. Primitive Compilation errors (26%) result from incomplete or mis-specified primitive sets that lead execution toward the wrong formalization. Orientation errors (34%), the largest category, show that grounding object-centric or event-conditioned directions remains the main bottleneck. API Tool errors (18%) arise from incorrect parameter passing, tool sequencing, unresolved dependencies, or budget exhaustion. Program errors (13%) are less frequent and are mostly caused by mistakes in deterministic geometric computation, such as missing coordinate transformations or incorrect metric mapping. Reconstruction errors (9%) reflect failures in the 3D scene geometry, where inaccurate poses or point clouds prevent reliable primitive instantiation.

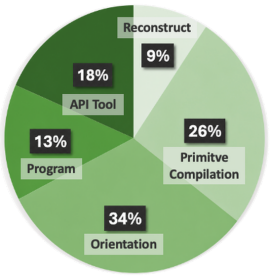


Figure 6: Failure cases distribution.

4 Related Work

Recent spatial reasoning methods move beyond direct VLM inference through external tools Wu et al. (2025b); Marsili et al. (2025), visual intermediate reasoning Wu et al. (2025a), and explicit 3D representations Fan et al. (2025); Zheng et al. (2025). Visual programming and agentic methods further decompose questions into executable code, tool plans, dynamic APIs, or geometric computation Gupta & Kembhavi (2023); Surís et al. (2023); Luo et al. (2026). While these approaches improve grounding and compositionality, their query-level artifacts often remain open-ended, mixing perception, planning, and answer derivation after the query is transformed into a program, plan, or formalization. A complementary line studies minimal sufficient evidence, showing that task-relevant observations can improve reliability and reduce distractors Guo et al. (2025). Together, these works suggest that spatial reasoning requires not only stronger tools or richer geometry, but tighter control over what information is committed before computation. See Appendix B for full related work.

5 Conclusion

We present PEPS, a training-free agentic framework that addresses under-committed query compilation in spatial reasoning by parsing queries into executable primitives under a typed FESM schema. By separating primitive parsing, geometric execution, and trace-level verification, PEPS makes reasoning commitments explicit, bounded, and diagnosable. Experiments across multi-image MMSI-Bench, viewpoint-sensitive ViewSpatial-Bench, and monocular OMNI3D-Bench show consistent improvements over strong VLM and tool-integrated baselines. More broadly, our results suggest that reliable spatial reasoning depends not only on stronger perception tools or richer 3D geometry, but also on explicitly committing to the right representational requirements before computation begins.

AI USE STATEMENT

In this work, we used generative AI tools solely for language editing and writing polish, including grammar checking, wording refinement, and readability improvements. We did not use generative AI tools to develop the research methodology, design experiments, interpret results, or formulate scientific claims. All AI-assisted edits were reviewed by the authors. We take full responsibility for the final content of this work.

REFERENCES

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. Qwen2.5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.

Ruth MJ Byrne and Philip N Johnson-Laird. Spatial reasoning. *Journal of memory and language*, 28 (5):564–575, 1989.

Zeren Chen, Xiaoya Lu, Zhijie Zheng, Pengrui Li, Lehan He, Yijin Zhou, Jing Shao, Bohan Zhuang, and Lu Sheng. Geometrically-constrained agent for spatial reasoning. *arXiv preprint arXiv:2511.22659*, 2025.

Zhe Chen, Weiyun Wang, Yue Cao, Yangzhou Liu, Zhangwei Gao, Erfei Cui, Jinguo Zhu, Shenglong Ye, Hao Tian, Zhaoyang Liu, et al. Expanding performance boundaries of open-source multimodal models with model, data, and test-time scaling. *arXiv preprint arXiv:2412.05271*, 2024.

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. *Advances in Neural Information Processing Systems*, 37:135062–135093, 2024.

Zhiwen Fan, Jian Zhang, Renjie Li, Junge Zhang, Runjin Chen, Hezhen Hu, Kevin Wang, Huaizhi Qu, Shijie Zhou, Dilin Wang, et al. Vlm-3r: Vision-language models augmented with instruction-aligned 3d reconstruction. *arXiv preprint arXiv:2505.20279*, 2025.

Gunnar Farnebäck. Two-frame motion estimation based on polynomial expansion. In *Scandinavian conference on Image analysis*, pp. 363–370. Springer, 2003.

Google DeepMind. Gemini 2.5: Our most intelligent ai model. `https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/`, 2025.

Yejie Guo, Yunzhong Hou, Wufei Ma, Meng Tang, and Ming-Hsuan Yang. Pursuing minimal sufficiency in spatial reasoning. *arXiv preprint arXiv:2510.16688*, 2025.

Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14953–14962, 2023.

Pascal Hitzler and Md Kamruzzaman Sarker. Neuro-symbolic artificial intelligence: The state of the art. 2022.

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

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

AI Jaided. Easyocr: Ready-to-use ocr with 80+ supported languages. *GitHub repository*, 2020.

Mengdi Jia, Zekun Qi, Shaochen Zhang, Wenyao Zhang, Xinqiang Yu, Jiawei He, He Wang, and Li Yi. Omnispatial: Towards comprehensive spatial reasoning benchmark for vision language models. *arXiv preprint arXiv:2506.03135*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Bo Li, Yuanhan Zhang, Dong Guo, Renrui Zhang, Feng Li, Hao Zhang, Kaichen Zhang, Peiyuan Zhang, Yanwei Li, Ziwei Liu, et al. Llava-onevision: Easy visual task transfer. *arXiv preprint arXiv:2408.03326*, 2024.

Dingming Li, Hongxing Li, Zixuan Wang, Yuchen Yan, Hang Zhang, Siqi Chen, Guiyang Hou, Shengpei Jiang, Wenqi Zhang, Yongliang Shen, et al. Viewspatial-bench: Evaluating multi-perspective spatial localization in vision-language models. *arXiv preprint arXiv:2505.21500*, 2025.

Zekun Li, Baolin Peng, Pengcheng He, Michel Galley, Jianfeng Gao, and Xifeng Yan. Guiding large language models via directional stimulus prompting. *Advances in Neural Information Processing Systems*, 36:62630–62656, 2023.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023.

Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Qing Jiang, Chunyuan Li, Jianwei Yang, Hang Su, et al. Grounding dino: Marrying dino with grounded pre-training for open-set object detection. In *European conference on computer vision*, pp. 38–55. Springer, 2024.

Zhanpeng Luo, Ce Zhang, Silong Yong, Cunxi Dai, Qianwei Wang, Haoxi Ran, Guanya Shi, Katia Sycara, and Yaqi Xie. pyspatial: Generating 3d visual programs for zero-shot spatial reasoning. *arXiv preprint arXiv:2603.00905*, 2026.

Wufei Ma, Luoxin Ye, Celso M de Melo, Alan Yuille, and Jieneng Chen. Spatialllm: A compound 3d-informed design towards spatially-intelligent large multimodal models. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 17249–17260, 2025.

Damiano Marsili, Rohun Agrawal, Yisong Yue, and Georgia Gkioxari. Visual agentic ai for spatial reasoning with a dynamic api. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 19446–19455, 2025.

Meta. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models. `https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/`, 2024.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pp. 561–577, 2018.

OpenAI. Hello gpt-4o, August 2024. URL `https://openai.com/index/hello-gpt-4o/`.

OpenAI. Introducing o3 and o4 mini. `https://openai.com/index/introducing-o3-and-o4-mini/`, 2025.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 3806–3824, 2023.

Qwen Team. Qwen3-vl: Sharper vision, deeper thought, broader action. `https://qwen.ai/blog?id=99f0335c4ad9ff6153e517418d48535ab6d8afef&from=research.latest-advancements-list`, 2025. Accessed: 2026-04-09.

Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, et al. Sam 2: Segment anything in images and videos. *arXiv preprint arXiv:2408.00714*, 2024.

Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 11888–11898, 2023.

Barbara Tversky. Spatial mental models. *Psychology of learning and motivation*, 27:109–145, 1991.

Jianyuan Wang, Minghao Chen, Nikita Karaev, Andrea Vedaldi, Christian Rupprecht, and David Novotny. Vggt: Visual geometry grounded transformer. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 5294–5306, 2025a.

Qianqian Wang, Yifei Zhang, Aleksander Holynski, Alexei A Efros, and Angjoo Kanazawa. Continuous 3d perception model with persistent state. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 10510–10522, 2025b.

Ruicheng Wang, Sicheng Xu, Yue Dong, Yu Deng, Jianfeng Xiang, Zelong Lv, Guangzhong Sun, Xin Tong, and Jiaolong Yang. Moge-2: Accurate monocular geometry with metric scale and sharp details. *arXiv preprint arXiv:2507.02546*, 2025c.

Yifan Wang, Jianjun Zhou, Haoyi Zhu, Wenzheng Chang, Yang Zhou, Zizun Li, Junyi Chen, Jiangmiao Pang, Chunhua Shen, and Tong He. pi^3: Permutation-equivariant visual geometry learning. *arXiv preprint arXiv:2507.13347*, 2025d.

Zehan Wang, Ziang Zhang, Tianyu Pang, Chao Du, Hengshuang Zhao, and Zhou Zhao. Orient anything: Learning robust object orientation estimation from rendering 3d models. *arXiv preprint arXiv:2412.18605*, 2024.

Junfei Wu, Jian Guan, Kaituo Feng, Qiang Liu, Shu Wu, Liang Wang, Wei Wu, and Tieniu Tan. Reinforcing spatial reasoning in vision-language models with interwoven thinking and visual drawing. *arXiv preprint arXiv:2506.09965*, 2025a.

Shengguang Wu, Xiaohan Wang, Yuhui Zhang, Hao Zhu, and Serena Yeung-Levy. Transductive visual programming: Evolving tool libraries from experience for spatial reasoning. *arXiv preprint arXiv:2512.20934*, 2025b.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

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. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 10632–10643, 2025b.

Sihan Yang, Runsen Xu, Yiman Xie, Sizhe Yang, Mo Li, Jingli Lin, Chenming Zhu, Xiaochen Chen, Haodong Duan, Xiangyu Yue, et al. Mmsi-bench: A benchmark for multi-image spatial intelligence. *arXiv preprint arXiv:2505.23764*, 2025c.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Satlm: Satisfiability-aided language models using declarative prompting. *Advances in Neural Information Processing Systems*, 36:45548–45580, 2023.

Baiqiao Yin, Qineng Wang, Pingyue Zhang, Jianshu Zhang, Kangrui Wang, Zihan Wang, Jieyu Zhang, Keshigeyan Chandrasegaran, Han Liu, Ranjay Krishna, et al. Spatial mental modeling from limited views. In *Structural Priors for Vision Workshop at ICCV'25*, 2025.

Songsong Yu, Yuxin Chen, Hao Ju, Lianjie Jia, Fuxi Zhang, Shaofei Huang, Yuhan Wu, Rundi Cui, Binghao Ran, Zaibin Zhang, et al. How far are vlms from visual spatial intelligence? a benchmark-driven perspective. *arXiv preprint arXiv:2509.18905*, 2025.

Duo Zheng, Shijia Huang, and Liwei Wang. Video-3d llm: Learning position-aware video representation for 3d scene understanding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8995–9006, 2025.

Chenming Zhu, Tai Wang, Wenwei Zhang, Jiangmiao Pang, and Xihui Liu. Llava-3d: A simple yet effective pathway to empowering lmms with 3d-awareness. *arXiv preprint arXiv:2409.18125*, 2024.

Jinguo Zhu, Weiyun Wang, Zhe Chen, Zhaoyang Liu, Shenglong Ye, Lixin Gu, Yuchen Duan, Hao Tian, Weijie Su, Jie Shao, et al. Internvl3: Exploring advanced training and test-time recipes for open-source multimodal models. *arXiv:2504.10479*, 2025.

A Implementation Details

A.1 Visual Foundation Models

The Executor instantiates the parsed primitives through a set of off-the-shelf visual foundation models. These models are used as perception backends for filling primitive values, such as camera pose, object location, object extent, metric scale, and orientation. The subsequent spatial relation is computed deterministically from the instantiated variables rather than inferred directly by these models.

- **VGGT** Wang et al. (2025a). We use VGGT to build the shared 3D workspace from the input image set. Given one or multiple views, it estimates camera parameters, depth maps, and point maps, which provide the common geometric substrate for grounding camera-based frames, object centroids, and cross-view spatial relations. In our pipeline, VGGT supplies the reconstruction context on which later primitive values are instantiated.
- **MoGe-2** Wang et al. (2025c). MoGe-2 is used when a query requires metric-scale reasoning. It predicts monocular geometry with scale information, allowing the relative 3D reconstruction to be aligned with real-world units. We use this scale estimate to support primitives and metrics involving physical size, distance, or measurements in meters.
- **GroundingDINO** Liu et al. (2024). We use GroundingDINO for open-vocabulary entity localization. Given textual entity descriptions from the parsed Entity primitives, it returns candidate boxes for objects, regions, or referring expressions in the specified view. These detections serve as the entry point for grounding anchors, targets, and candidate sets before acquiring their 3D states.
- **SAM-2** Ravi et al. (2024). SAM-2 refines detected boxes into object-level masks. We use these masks to select the corresponding 3D points from the reconstructed workspace, thereby converting 2D detections into entity-level point sets. This bridge is needed for state primitives such as point clouds, centroids, and 3D extents.
- **Orient Anything** Wang et al. (2024). Orient Anything provides object orientation estimates from a single view. We use its predicted pose or principal direction to instantiate orientation-related State primitives and to construct object-based reference frames when a query depends on an object's facing direction, surface normal, or local coordinate system.

A.2 Tool Interfaces

- `reconstruct`. This API builds the shared 3D workspace from the input image set. Given one or multiple views, it invokes VGGT to estimate camera intrinsics, camera extrinsics, depth, and world-space point maps. For multi-view inputs with possible scene variation, the pipeline first identifies stable overlapping regions or common static objects for alignment before constructing the unified reconstruction. The resulting workspace provides the geometric substrate for camera-based frames, cross-view grounding, and object-level state acquisition.
- `detect`. This API localizes entities specified by the parsed Entity primitives. It takes a text description and a target view as input, and returns candidate bounding boxes with labels. Depending on the deployed backbone, localization is performed either through a capable VLM prompted for region grounding or through GroundingDINO for open-vocabulary detection.
- `project_box_to_3d_points`. This API converts a localized 2D entity into a 3D entity state. Starting from a detected bounding box, it obtains an object mask with SAM-2 and uses the mask to select corresponding points from the reconstructed world-space point map. The output is an entity-level point set, from which centroids, extents, and surface-level quantities can be derived.
- `predict_obj_pose`. This API estimates the local pose of an object when a query requires orientation-sensitive reasoning. It combines the object's 3D location from the projected point set with an orientation estimate from Orient Anything, and returns an object-to-world transform. The transform is used to instantiate orientation states and object-based reference frames.
- `estimate_scale`. This API is invoked for queries involving metric distance, physical size, or real-world units. It aligns the relative reconstruction produced by VGGT with the metric geometry

estimated by MoGe-2, producing a global scale factor that maps reconstructed coordinates to meters.

- `ocr`. This API recognizes text in an image and returns detected strings with their bounding regions. We use it when textual cues are part of entity grounding, region filtering, or answer candidate identification.
- `analyze_motion`. This API estimates image-level motion between sequential views using Farneback optical flow Farnebäck (2003). It is used for motion-sensitive queries where small viewpoint changes or local camera displacement may not be reliably captured by full 3D reconstruction alone.
- `code`. This API performs the final deterministic computation over the instantiated primitive values. It receives the filled workspace variables, the retrieved FESM requirements, and the relevant geometric rules, then generates and executes sandboxed Python code. The code is constrained to use only available variables, preventing the computation stage from inventing new perceptual evidence or revising the committed requirements.

A.3 Agentic Framework

We implement the system as a modular pipeline that separates VLM-based reasoning from perception and geometric computation. This separation allows the Parser, Executor, and Verifier to share a common state while invoking visual tools only when required by the current FESM commitments.

State and Orchestration. The pipeline is implemented with Ray Moritz et al. (2018) and LangGraph. LangGraph maintains the intermediate workspace, including parsed primitives, tool outputs, execution traces, and verifier feedback, and coordinates the iterative loop.

Tool Deployment. Perception and computation modules are served as independent Ray Serve actors. This design supports parallel tool calls, such as reconstruction and detection, and isolates failures from individual visual backends. In our experiments, the tool suite runs on 2 NVIDIA A100 GPUs.

VLM Deployment. VLM roles are deployed separately from the tool services. Open-source backbones are served with vLLM Kwon et al. (2023) on 4 NVIDIA A100 GPUs, while closed-source models are accessed through their official APIs. We use role-specific decoding: the Parser, Executor, and Verifier use `TEMPERATURE=0.6` and `TOP_P=0.95` to allow flexible requirement analysis and feedback, whereas the code generation stage uses `TEMPERATURE=0.0` for deterministic computation.

A.4 Implementation

Agent Configuration. Unless otherwise specified, the Parser, Executor, and Verifier are instantiated with GPT-4o. For open-source backbone experiments, we replace the VLM used by all agent roles with the evaluated backbone while keeping the tool suite, prompts, and execution protocol unchanged. We use `TEMPERATURE=0.6` and `TOP_P=0.95` for requirement parsing, tool orchestration, and verification, and set `TEMPERATURE=0.0` for code generation to reduce stochastic variation in geometric computation. Each query is allowed at most 3 parsing–execution–verification rounds.

Schema and Example Library. The initial FESM schema is induced from 300 sampled training/development questions, with 100 questions from each source benchmark. Candidate primitives are merged by semantic equivalence and retained only when they are supported by at least 3 questions and can be grounded by the available tool interface. During execution, verifier feedback is categorized as either *missing-slot* or *primitive-gap*. Missing-slot feedback triggers re-parsing within the existing schema, while primitive-gap feedback adds a temporary primitive candidate to the pool. A candidate primitive is promoted to the schema after being reused in at least 5 accepted cases with verifier score above $\tau = 0.7$. The Example Library stores accepted traces, including the query, parsed primitives, instantiated variables, generated program, final answer, and verifier feedback. For each new query, we retrieve the top-3 examples by embedding similarity and use them as in-context demonstrations. Ground-truth answers from test sets are never used for schema induction, example retention, or verifier feedback.

Execution Protocol. The Executor issues tool calls only to instantiate unresolved FESM requirements. Each query has a maximum budget of 12 perception/tool calls and 3 code-generation attempts.

Reconstruction is cached per input scene, and entity detections are cached by view and text prompt. If a tool returns multiple candidates, the Executor applies the parsed view, region, attribute, and cardinality constraints for disambiguation; if no valid candidate remains, the case is marked as unresolved and returned to the Verifier. Parallelizable calls, such as detecting multiple entities after reconstruction, are executed concurrently.

Code Execution and Answer Normalization. Generated Python code runs in a sandbox with access only to the filled workspace variables and standard numerical libraries such as `numpy`, `scipy`, and `math`. The code stage cannot request new perception results, introduce undeclared variables, or modify the parsed requirements. Each execution is limited to 10 seconds. For multiple-choice questions, the output is mapped to the closest valid option; for yes/no questions, it is normalized to a binary label; for directional answers, synonyms are canonicalized to the benchmark label set; and for numerical answers, units and rounding are normalized according to the official evaluation protocol.

Evaluation Setup. All benchmarks are converted to a unified input format consisting of images, view identifiers, question text, and optional answer candidates. Multi-image inputs preserve their original view order, while single-image 3D tasks are treated as one-view scenes. We use the original benchmark questions without ground-truth intermediate supervision. Baselines are evaluated using their official code and recommended settings whenever available; otherwise, we follow the closest released prompt and inference configuration. All experiments are run on NVIDIA A100 GPUs with the perception tool suite deployed separately from the VLM inference service.

B Related Work

Spatial Reasoning in Vision-Language Models. Spatial reasoning requires models to interpret object relations, reference frames, viewpoint changes, and 3D scene structure, and remains a persistent challenge for modern VLMs Yang et al. (2025b); Yu et al. (2025); Yang et al. (2025c); Li et al. (2025); Yin et al. (2025); Jia et al. (2025). Recent work improves spatial ability through stronger multimodal backbones, spatial benchmarks, specialized instruction tuning, and geometry-aware representations Hurst et al. (2024); OpenAI (2024); Li et al. (2024); Bai et al. (2025); Qwen Team (2025); Cheng et al. (2024); Ma et al. (2025). These approaches narrow the gap between 2D visual recognition and 3D spatial understanding by injecting depth, reconstruction, position-aware features, or embodied observations. However, learned spatial representations alone do not necessarily expose whether the model has selected the correct viewpoint, grounded the intended entities, or acquired the geometric variables needed by the query.

Tool-Integrated and Geometric Spatial Reasoning. Another line of work moves spatial reasoning beyond direct VLM inference by introducing executable intermediate representations, external perception tools, dynamic APIs, and deterministic geometric computation Gupta & Kembhavi (2023); Surís et al. (2023); Wu et al. (2025b); Marsili et al. (2025); Luo et al. (2026); Chen et al. (2025); Fan et al. (2025). Program- and agent-based systems improve compositionality by translating visual questions into code, tool plans, or modular reasoning traces. 3D-geometric systems further ground inference with reconstructed scenes, camera poses, object-level geometry, and metric computation. While these methods reduce reliance on unconstrained semantic reasoning, their generated artifacts often remain open-ended, mixing perception, planning, and answer derivation in one output space. Thus, plausible execution traces may still hide missing spatial commitments, such as the intended reference frame, entity constraint, state variable, or relation metric.

Evidence Selection and Constrained Reasoning. A complementary direction studies task-relevant evidence and minimal sufficiency, showing that spatial reasoning can benefit from selecting only the observations necessary for the query Guo et al. (2025). More broadly, constrained reasoning restricts model behavior by translating natural language into explicit variables, constraints, formal programs, or verifiable symbolic structures before solving Hitzler & Sarker (2022); Pan et al. (2023); Ye et al. (2023); Liu et al. (2023); Li et al. (2023); Huang et al. (2024). Related ideas have been explored in logical reasoning, planning, and embodied manipulation, where language models serve as translators into formal representations that can be checked or optimized by external solvers. Together, these works suggest that reliable spatial reasoning depends not only on stronger tools or richer geometry, but also on tighter control over the early query-to-requirement step before downstream execution.

C Evaluation Benchmark Details

C.1 Benchmark Details

C.1.1 MMSI-Bench

MMSI-Bench Yang et al. (2025c) evaluates multi-image spatial intelligence in real-world scenes. Each question is paired with multiple images captured from different viewpoints, requiring the model to integrate cross-view evidence rather than reason from a single observation. This benchmark is particularly relevant to our setting because many questions depend on resolving view-specific entities, camera viewpoints, and spatial relations that are only partially observable in each image. It contains four major categories:

- **Positional Relationship.** This category tests whether a model can determine relative spatial relations among objects, cameras, and semantic regions across views. Typical questions require identifying where one entity is located with respect to another after grounding both entities in the multi-view scene.
- **Attribute.** This category focuses on spatially related object attributes, including shape, size, length, and other geometry-dependent properties. Unlike simple recognition tasks, these questions often require comparing or estimating physical properties from visual evidence.
- **Motion.** This category evaluates reasoning about object or camera movement. The model must infer how a viewpoint or entity changes across images, which requires consistent alignment of observations over time or across views.
- **Multi-Step Reasoning.** This category combines multiple spatial operations into a longer reasoning chain. Questions may require first grounding entities in different views, then resolving a reference frame, and finally computing or comparing a relation.

C.1.2 ViewSpatial-Bench

ViewSpatial-Bench Li et al. (2025) is designed to evaluate viewpoint-sensitive spatial localization. Its questions emphasize that spatial terms such as left, right, front, and behind are not absolute scene properties, but depend on the viewpoint or agent-centered frame under which the relation is interpreted. This benchmark is useful for testing whether a model can distinguish externally anchored camera frames from human-centered or embodied perspectives.

- **Camera-Based.** This split evaluates spatial reasoning from the perspective of a specified camera or image view. The model must interpret directional relations according to the selected viewpoint, rather than relying on an implicit or default image frame.
- **Person-Based.** This split evaluates reasoning from a person-centered frame. The queried relation must be interpreted with respect to the orientation or position of a human observer, making it especially sensitive to frame grounding and egocentric direction understanding.

C.1.3 OMNI3D-Bench

OMNI3D-Bench Marsili et al. (2025) evaluates fine-grained 3D spatial reasoning from single images. Compared with multi-view benchmarks, this setting is more challenging in terms of depth ambiguity and metric estimation because the model must infer 3D layout from monocular visual evidence. The benchmark includes different answer formats, covering both categorical and numerical spatial reasoning.

- **Counting.** This category tests whether a model can count spatially relevant entities under the constraints specified by the question. It often requires grounding a candidate set and filtering objects according to visual or spatial conditions.
- **Float-MRA.** This category evaluates numerical 3D reasoning, such as estimating distances, sizes, or other continuous geometric quantities. Following the benchmark protocol, performance is measured using mean relative accuracy rather than exact-match accuracy.

- **Yes/No.** This category asks binary spatial questions, requiring the model to verify whether a queried relation or geometric condition holds in the scene.
- **Multiple Choice.** This category provides a fixed set of candidate answers. The model must select the option best supported by the inferred 3D relation, object layout, or metric property.

D LIMITATIONS

PEPS reduces under-committed query compilation through a closed FESM schema, but this only shifts the failure mode rather than eliminating it. The library remains inherently bounded, and the conservative primitive-promotion rule means truly novel spatial concepts are under-served until sufficient evidence accumulates. The framework is also upper-bounded by its perception backbones: the failure analysis attributes most residual errors to orientation grounding, tool-workflow issues, and 3D reconstruction rather than to compilation itself, and the iteration-budget ablation confirms that the corrective loop cannot repair perception noise. Finally, the agent roles depend on a strong proprietary backbone, and the FESM schema is induced from a narrow slice of static-scene benchmarks, leaving embodied, dynamic, and out-of-distribution settings untested.

E FUTURE WORK

Two directions follow naturally. First, making schema growth more automatic, for example through learned promotion criteria, would let the library evolve faster and extend more gracefully to new task families. Second, since the bottleneck has shifted from parsing to perception, the typed traces produced by PEPS could in turn be used to supervise or distill the perception backbones themselves, and to train smaller open-source agents that reduce dependence on proprietary APIs. Extending FESM with temporal and embodied primitives would broaden coverage to video, manipulation, and navigation, and the verifier's compilation-success signal could be developed into an explicit confidence score for selective prediction or human hand-off.

F QUALITATIVE EXAMPLE

Input Metadata


```
{
  "image_index": "86379.887_00000637.jpg",
  "question_index": 475,
  "image_filename": "ARKitScenes/Training/42446574/86379.887_00000637.jpg",
  "question": "If the bowl on the table has a 3D diameter of 4 meters, how long is the table in 3D in meters?",
  "answer": 14.0,
  "answer_type": "float"
}
```




Parser Agent

```
Input:
  Query:
    If the bowl on the table has a 3D diameter of 4 meters,
    how long is the table in 3D in meters?

Parsed FESM Primitives:

Frame:
  frame_type = none

Entity:
  anchor = bowl
  target = table
  view_source(anchor) = all_views
  view_source(target) = all_views
  count(anchor) = one
  count(target) = one
  filter(anchor) = on the table

State:
  bbox3d(bowl, oriented)
  bbox3d(table, oriented)

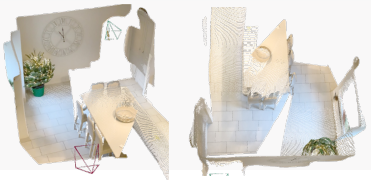
Metric:
  size_comparison(table, bowl, axis = length/diameter)
```

```
The query asks for the metric length of the table, using the bowl's known
3D diameter as a scale reference. It does not involve left, right, front,
behind, or viewpoint-dependent relations, so no reference frame is needed.

The bowl is the reference entity because its real-world diameter is given.
The table is the target entity because its 3D length is being queried.
The phrase "bowl on the table" constrains the bowl grounding.

Both entities require oriented 3D bounding boxes. The bowl needs an estimated
3D diameter in reconstruction units, and the table needs an estimated 3D
length in the same reconstruction units. The final metric compares these two
size quantities and transfers the known metric scale from the bowl to the table.
```

3D Reconstruction



Executor Agent

```
Execution Trace:

[Frame Primitive]
frame_type = none
  -> skip frame construction
  -> no camera-based, object-based, or direction-based frame is required

[Entity Primitive]
anchor = bowl
filter(anchor) = on the table
view_source(anchor) = all_views
count(anchor) = one
  -> detect("bowl on the table", views=all_views)
  -> output: workspace/detections/bowl_box.json
  -> filled primitive: anchor = bowl

[Entity Primitive]
target = table
view_source(target) = all_views
count(target) = one
  -> detect("table", views=all_views)
  -> output: workspace/detections/table_box.json
  -> filled primitive: target = table

[State Primitive]
bbox3d(bowl, oriented)
  -> reconstruct(["86379.887_00000637.jpg", "86425.885_00003398.jpg"])
  -> output:
       workspace/recon/scene_3d.glb
       workspace/recon/cameras.json

  -> project_box_to_3d_points(
       workspace/detections/bowl_box.json,
       workspace/recon/scene_3d.glb
     )
  -> output:
       workspace/points/bowl_points.ply

  -> fit_oriented_bbox(workspace/points/bowl_points.ply)
  -> output:
       bowl_obb = {
         "diameter": 0.286
       }

[State Primitive]
bbox3d(table, oriented)
  -> project_box_to_3d_points(
       workspace/detections/table_box.json,
       workspace/recon/scene_3d.glb
     )
  -> output:
       workspace/points/table_points.ply

  -> fit_oriented_bbox(workspace/points/table_points.
  -> output:
       table_obb = {
         "length": 1.002
       }

[Metric Primitive]
size_comparison(table, bowl, axis = length/diameter)
  -> code(workspace)
  -> input workspace:
       {
         "known_bowl_diameter": 4.0,
         "bowl_obb": {
           "diameter": 0.286
         },
         "table_obb": {
           "length": 1.002
         }
       }
```

Workspace


```
{
  "scene_3d": "workspace/recon/scene_3d.glb",
  "cameras": "workspace/recon/cameras.json",
  "bowl_box": "workspace/detections/bowl_box.json",
  "table_box": "workspace/detections/table_box.json",
  "bowl_pointset": "workspace/points/bowl_points.ply",
  "table_pointset": "workspace/points/table_points.ply",
  "known_bowl_diameter": 4.0,
  "bowl_obb": {
    "diameter": 0.286
  },
  "table_obb": {
    "length": 1.002
  }
}
```


Code Generation

```
def execute(workspace):
    table_length_recon = workspace["table_obb"]["length"]
    bowl_diameter_recon = workspace["bowl_obb"]["diameter"]
    known_bowl_diameter = workspace["known_bowl_diameter"]

    scale = known_bowl_diameter / bowl_diameter_recon
    table_length_meters = table_length_recon * scale

    return round(float(table_length_meters), 1)
```

Deterministic Computation

```
Given:
  known_bowl_diameter = 4.0 meters
  bowl_diameter_recon = 0.286
  table_length_recon = 1.002

Compute scale:
  scale = known_bowl_diameter / bowl_diameter_recon
        = 4.0 / 0.286
        = 13.986

Compute table length:
  table_length_meters = table_length_recon * scale
                      = 1.002 * 13.986
                      = 14.014

Round:
  answer = round(14.014, 1)
         = 14.0

Final Answer:
  14.0 meters
```

Verifier Agent

```
<verification_decision>accept</verification_decision>

<quality_score>9.2</quality_score>

<reasoning>
The query requires only metric size scaling, so frame_type = none is valid.
The trace grounds the correct bowl on the table and the target table, extracts
their oriented 3D boxes, and uses the given bowl diameter of 4.0 meters to
scale the reconstructed table length. The final computation,
1.002 * (4.0 / 0.286) = 14.0, directly supports the answer.
</reasoning>

<feedback_type>none</feedback_type>

<feedback>none</feedback>
```

Figure 7: Qualitative Example 1.

Input Metadata


```
{
  "image_index": "31920.525_00003464.jpg",
  "question_index": 451,
  "image_filename": "ARKitScenes/Training/42444574/31920.525_00003464.jpg",
  "input_images": [
    "omni3d-bench/images/ARKitScenes/Training/42444574/31911.512_00002923.jpg",
    "omni3d-bench/images/ARKitScenes/Training/42444574/31920.525_00003464.jpg",
    "omni3d-bench/images/ARKitScenes/Training/42444574/31936.519_00004424.jpg"
  ],
  "question": "Is the ratio of chairs to lamps less than 2?",
  "ground_truth": "no",
  "answer_type": "str"
}
```




Parser Agent

```
Input Query:
  Is the ratio of chairs to lamps less than 2?

Parsed FESM Primitives:

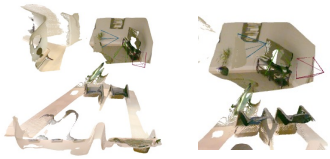
Frame:
  frame_type = none

Entity:
  candidate = open_set_detection{chairs}
  candidate = open_set_detection{lamps}
  view_source(chairs) = all_views
  view_source(lamps) = all_views
  count(chairs) = multiple
  count(lamps) = multiple

State:
  centroid(chair_instances)
  centroid(lamp_instances)

Metric:
  count_metric(candidate_entity_set = chairs)
  count_metric(candidate_entity_set = lamps)
  ratio_comparison(
    numerator = count(chairs),
    denominator = count(lamps),
    threshold = 2,
    operator = less_than
  )
```

3D Reconstruction



Code Generation

```
def execute(workspace):
    chair_count = int(workspace["chair_count"])
    lamp_count = int(workspace["lamp_count"])
    threshold = float(workspace["threshold"])

    if lamp_count == 0:
        raise ValueError("lamp_count is zero.")

    ratio = chair_count / lamp_count
    is_less_than_threshold = ratio < threshold

    return {
        "ratio": float(ratio),
        "answer": "yes" if is_less_than_threshold else "no"
    }
```

Verifier Agent

```
<verification_decision>accept</verification_decision>

<quality_score>9.1</quality_score>

<reasoning>
The query only requires counting chairs and lamps and comparing their ratio,
so frame_type = none is valid.

The trace detects both categories across all views and uses 3D association
to avoid duplicate cross-view counts.

It obtains four chair instances and one lamp instance.

The computation 4 / 1 = 4 is not less than 2,
so the answer "no" is directly supported.
</reasoning>

<feedback_type>none</feedback_type>

<feedback>none</feedback>
```

Executor Agent

```
Execution Trace:

[Frame Primitive]
frame_type = none
  -> skip frame construction
  -> the query is frame-agnostic because it asks for an object-count ratio

[Entity Primitive]
candidate = open_set_detection{chairs}
view_source(chairs) = all_views
count(chairs) = multiple
  -> detect("chair", views=[
       "31911.512_00002923.jpg",
       "31920.525_00003464.jpg",
       "31936.519_00004424.jpg"
     ])
  -> output:
       workspace/detections/chair_boxes_view1.json
       workspace/detections/chair_boxes_view2.json
       workspace/detections/chair_boxes_view3.json

[Entity Primitive]
candidate = open_set_detection{lamps}
view_source(lamps) = all_views
count(lamps) = multiple
  -> detect("lamp", views=[
       "31911.512_00002923.jpg",
       "31920.525_00003464.jpg",
       "31936.519_00004424.jpg"
     ])
  -> output:
       workspace/detections/lamp_boxes_view1.json
       workspace/detections/lamp_boxes_view2.json
       workspace/detections/lamp_boxes_view3.json

[State Primitive]
centroid(chair_instances)
centroid(lamp_instances)
  -> reconstruct([
       "31911.512_00002923.jpg",
       "31920.525_00003464.jpg",
       "31936.519_00004424.jpg"
     ])
  -> output:
       workspace/recon/scene_3d.glb
       workspace/recon/cameras.json

  -> project_box_to_3d_points(chair_boxes, scene_3d)
  -> output:
       workspace/points/chair_points_view1.ply
       workspace/points/chair_points_view2.ply
       workspace/points/chair_points_view3.ply

  -> project_box_to_3d_points(lamp_boxes, scene_3d)
  -> output:
       workspace/points/lamp_points_view1.ply
       workspace/points/lamp_points_view2.ply
       workspace/points/lamp_points_view3.ply

  -> cross_view_associate_by_3d_centroid(chair_points)
  -> output:
       chair_instances = [
         "chair_1",
         "chair_2",
         "chair_3",
         "chair_4"
       ]
       count(chairs) = 4

  -> cross_view_associate_by_3d_centroid(lamp_points)
  -> output:
       lamp_instances = [
         "lamp_1"
       ]
       count(lamps) = 1

[Metric Primitive]
count_metric(candidate_entity_set = chairs)
count_metric(candidate_entity_set = lamps)
ratio_comparison(count(chairs), count(lamps), threshold = 2, operator = less_than)
  -> code(workspace)
  -> input workspace:
       {
         "chair_count": 4,
         "lamp_count": 1,
         "threshold": 2,
         "operator": "less_than"
       }
```

Figure 8: Qualitative Example 2.

G DETAILED FESM PRIMITIVE SCHEMA

Table 10: **FESM Schema: Frame Primitives.**

[Frame]
Purpose: These primitives determine which coordinate frame the query should be interpreted in. In spatial reasoning, the same object pair can lead to different answers under different frames. Therefore, before checking any entity state or relation metric, the system must first resolve the reference frame: *whose viewpoint is used*, *where the origin is*, and *which direction counts as forward/up*.

- **frame_type**: This primitive decides what kind of reference frame the query is asking for. It is the top-level choice that determines how the rest of the frame should be constructed, and it also uniquely determines the origin of the reference frame (i.e., the camera center for `camera_based`, the anchor entity's representative point for `object_based`, and the source entity's representative point for `direction_based`). The valid values:
  - `[camera_based]`: use the viewpoint of a specified image or camera.
  - `[object_based]`: use the frame defined by an anchor object's pose, orientation, or interaction semantics.
  - `[direction_based]`: use a frame whose forward direction is induced by an object-to-object direction or an object-to-cardinal alignment.
  - `[none]`: use when the query is frame-agnostic (e.g., counting, attribute, or scene-intrinsic queries) and no reference frame needs to be established. When selected, all other frame primitives are skipped.

  Example: `frame_type=camera_based`, `frame_type=object_based`.

- **frame_view**: This primitive decides which image/viewpoint is the reference viewpoint when `frame_type=camera_based`. It tells the system whose camera frame should be used to interpret left/right/front/behind. Example: `frame_view=view_1`, `frame_view=image_3`.

- **frame_anchor**: This primitive decides which object defines the reference frame when `frame_type=object_based`. In other words, it identifies the object whose geometry, orientation, or interaction role provides the spatial anchor. Example: `frame_anchor=sofa`, `frame_anchor=sink`.

- **frame_source / frame_target**: These two primitives decide which ordered object pair defines the reference direction. They are used when `frame_type=direction_based`, i.e., the frame is not taken from a camera or a single object, but from a directed relation between two objects. Conceptually, the forward direction is induced by the vector from source to target.
  - Example: in "oven is north of sink", `frame_source=sink` and `frame_target=oven`.

- **frame_forward**: This primitive decides what counts as the forward direction (`+Z_frame`) in the reference frame. This is one of the most important frame primitives, because front/back/left/right all depend on how the positive forward axis is defined. The valid values include:
  - `[+Z_cam]`: the camera view's front when `frame_type=camera_based`.
  - `[+Z_obj]`: the anchor object's semantic front axis when `frame_type=object_based`.
  - `[V_source→target]`: set to `normalize(Centroid_target − Centroid_source)` when `frame_type=direction_based`.

- **frame_down**: This primitive decides how the vertical axis (i.e., the positive down direction `+Y_frame`, following the OpenCV convention) of the reference frame is determined. Unlike the forward direction, the vertical axis cannot always be inherited from a single anchor, because the reconstructed world frame is not guaranteed to be gravity-aligned. The valid values include:
  - `[+Y_cam]`: follow the anchor camera's down axis when `frame_type=camera_based`.
  - `[+Y_obj]`: follow the anchor object's semantic vertical axis when `frame_type=object_based`.
  - `[gravity]`: estimate the gravity direction from scene cues (e.g., detected ground plane, wall normals, or an auxiliary gravity-estimation tool). This value is required when `frame_type=direction_based` or when the query explicitly involves gravity-aligned concepts (e.g., *above*, *below*, *top-down view*) and the anchor alone cannot provide a reliable vertical direction.

Table 11: **FESM Schema: Entity Primitives.**

[Entity]
Purpose: These primitives determine which scene entities the query is actually talking about. Before any geometric state or relation metric can be checked, the system must first ground the language in the query to concrete scene instances. These primitives resolve *who is the reference entity*, *who is the target*, *where they should be grounded from*, and *how ambiguous mentions should be narrowed down*.

- **anchor**: This primitive decides which entity serves as the reference in the query. It identifies the entity relative to which another entity, direction, or relation will be interpreted. Its value is typically an entity category or a resolved entity mention extracted from the query. Note that the `anchor` is not necessarily the same as `frame_anchor_entity` in the Frame primitives: the former is the relational reference for the queried metric, while the latter defines the coordinate frame. A single query may involve both, and they may or may not coincide.
  - Example: `anchor=sofa`, `anchor=sink`.

- **target**: This primitive decides which entity is the actual queried target. It identifies the entity whose location, orientation, or spatial relation is being asked about. Its value typically corresponds to a single entity mention in the query.
  - Example: `target=round dinner table`, `target=blue bin`.

- **candidate**: This primitive decides which set of entities should be considered as possible answers. It is required when the query does not ask about one fixed target, but instead asks the system to search over a set of candidate entities (commonly triggered by queries such as "which object . . . " or "what is . . . "). To distinguish how the candidate set is constructed, this primitive takes two modes:
  - `[enumerated]`: the candidates are explicitly listed in the query itself (e.g., as multiple-choice options), and the set is taken directly from the query text.
  - `[open_set_detection]`: the candidates are specified only by a category prompt (e.g., "all furniture", "any chair in the room"), and the set must be populated by open-vocabulary detection.
  - Example: `candidate=[enumerated]{table, painting, chair, desk}`, `candidate=[open_set_detection]{all chairs}`.

- **view_source**: This primitive decides from which input view(s) each entity should be grounded. It is essential for multi-view queries, where the same entity category may appear in multiple images and the query explicitly anchors an entity to a specific view (e.g., "the chair in image 3"). Without this primitive, downstream detection cannot know which image to search in, and cross-view confusion becomes a major failure mode. The valid values include:
  - `[view_id]`: a specific view index bound to the corresponding entity (e.g., `view_1`, `image_3`), used when the query ties the entity to one view.
  - `[all_views]`: ground the entity across all available views, used when the query is view-agnostic or when cross-view association is needed.
  - Example: `view_source(target_entity)=view_3`.

- **filter**: This primitive decides which scene region, local area, or contextual location the entity must belong to. It is needed when category, attribute, or view constraints alone are insufficient to uniquely identify the intended entity (e.g., "the painting *on the wall in the fitness area*"). Its value specifies a contextual localization constraint attached to a specific entity, to be consumed by downstream grounding as a disambiguation filter.
  - Example: `filter(target_entity)={on the wall, in the fitness area}`.

- **count**: This decides how many instances of a given entity the query expects to ground. It provides a cardinality constraint that the downstream grounding stage must enforce, and it is the primary trigger for ambiguity resolution: if the expected count does not match the actual number of detections, the system must invoke a disambiguation step (e.g., VLM-based re-selection using `entity_region_filter` or other cues) before proceeding. The valid values include:
  - `[one]`: exactly one instance is expected (e.g., "the sofa"); if multiple are detected, disambiguation is mandatory.
  - `[multiple]`: more than one instance is expected (e.g., "the chairs around the table"); the grounding should preserve all matching instances.
  - `[unspecified]`: the query does not impose any cardinality constraint, and any number of grounded instances is acceptable.
  - Example: `count(target)=one`.

Table 12: **FESM Schema: State Primitives.**

[State]
Purpose: These primitives determine which geometric states of the resolved cameras, entities, and scene must be acquired before any relation metric can be computed. The system should always follow the *coarse-first, fine-on-demand* principle: retrieve the simplest state that is sufficient for the downstream metric, and only retrieve a finer state when the metric explicitly requires it (e.g., retrieve `centroid` instead of `pointset` unless surface-level geometry is needed; retrieve `orientation` instead of the full `3Dpose` unless full 6-DoF is required).

- **camera_pose(view)**: Returns the 3D pose of the specified camera view in the world frame, including camera position and orientation. This state is required for camera-based frames and for any view-dependent relation such as depth, visibility, or novel-view reasoning. It is typically provided directly by the output of the `reconstruct` tool.
  - Example: `camera_pose(view_1)`.

- **centroid(entity)**: Returns the representative 3D center of an entity as a single 3D point in the world frame. This is the coarsest and most frequently used location state. It should be the default choice whenever the downstream metric only depends on position (e.g., displacement, direction, centroid distance), without requiring the entity's full shape or orientation.
  - Example: `centroid(chair)`.

- **bbox3d(entity, mode)**: Returns a 3D bounding box describing the entity's occupied spatial extent. This state serves two purposes: it provides a compact proxy for entity size and coarse geometry, and it directly supports size-comparison and coarse-overlap queries. The `mode` argument specifies whether the box is axis-aligned to the world frame or aligned to the entity's own semantic axes, which yields different notions of "size". The valid values include:
  - `[axis_aligned]`: an axis-aligned bounding box (AABB) in the world frame; suitable for rough spatial extent and coarse overlap.
  - `[oriented]`: an oriented bounding box (OBB) aligned with the entity's semantic axes; required when the query asks about width/length/height along the entity's own axes (e.g., "how long is the sofa"). The oriented mode implicitly depends on `entity_pose`.
  - Example: `bbox3d(refrigerator, axis_aligned)`, `bbox3d(sofa, oriented)`.

- **pointset(entity)**: Returns the full 3D point set associated with the entity as an entity-level 3D point cloud in the world frame. This is the finest location-level state and should only be retrieved when a centroid or bounding box is insufficient—for example, when the downstream metric requires surface-to-surface distance, fine-grained occupancy, contact reasoning, or region-based overlap.
  - Example: `pointset(table)`.

- **3Dpose(entity)**: Returns the full 6-DoF pose of the entity in the world frame (position + rotation). This is the finest orientation-level state and should only be retrieved when the query requires the entity's complete local coordinate system, most notably when the entity itself defines an object-based reference frame (i.e., `frame_type=object_based` with `frame_anchor_entity=entity`), or when a local-frame transformation is needed.
  - Example: `3Dpose(fireplace)`.

- **orientation(entity)**: Returns a single 3D axis vector describing the entity's primary directional axis. This state is the lightweight alternative to `3Dpose` when only one direction is needed rather than the full 6-DoF. It automatically dispatches based on the entity's geometric type:
  - For volumetric entities (e.g., sofa, chair, appliance), it returns the entity's semantic front axis (i.e., the direction it is "facing").
  - For planar entities (e.g., painting, door, monitor, wall), it returns the outward surface normal, which plays the same role as the front axis for planar objects.
  - Retrieving both `orientation` and `3Dpose` for the same entity is redundant; the coarse-first principle requires the retriever to pick only one.
  - Example: `orientation(sofa)`, `orientation(painting)`.

Table 13: **FESM Schema: Metric Primitives.**

[Metric]
Purpose: These primitives specify the geometric relation or quantity to be computed once the relevant frame, entities, and states are available. A query typically requires one top-level metric that directly answers the question, while other metrics may serve as intermediate quantities.

- **relative_displacement(entity_a, entity_b, frame)**: Returns the 3D displacement vector from `entity_a` to `entity_b` in the reference frame. Core metric for directional spatial relations. Computed from `entity_centroid` of both entities.
  - Example: `relative_displacement(chair, blue_bin, frame)`.
- **planar_displacement(entity_a, entity_b, frame)**: Returns the horizontal-plane displacement with the vertical component removed. Used for left/right/front/back and cardinal-direction reasoning. Derived from `relative_displacement`.
  - Example: `planar_displacement(sofa, table, frame)`.
- **vertical_offset(entity_a, entity_b, frame)**: Returns the signed height difference along the frame's vertical axis. Required for above/below queries. Computed from `entity_centroid` of both entities.
  - Example: `vertical_offset(counter, microwave, frame)`.
- **euclidean_distance(entity_a, entity_b)**: Returns the Euclidean distance between entity centroids. Frame-invariant. Used for near/far reasoning; when metric units are required, `estimate_scale` is additionally invoked. Example: `euclidean_distance(chair, desk)`.
- **surface_distance(entity_a, entity_b)**: Returns the minimum surface-to-surface distance between two entities, more reliable than centroid distance when entity extent matters. Computed from `entity_pointset` of both entities. Example: `surface_distance(sofa, coffee_table)`.
- **size_comparison(entity_a, entity_b, axis)**: Returns the signed size difference along a specified axis (entity-local or world). Used for bigger/longer/taller queries. Computed from `entity_bbox3d` of both entities. Example: `size_comparison(sofa, table, length)`.

- **orientation_difference(axis_1, axis_2)**: Returns the angular difference between two orientation axes. Used for alignment analysis. Inputs typically come from `entity_orientation_axis` or a frame axis Example: `orientation_difference(entity_orientation_axis(chair), entity_orientation_axis(table))`.
- **facing_alignment(entity, target, frame)**: Returns a signed scalar in $[-1, 1]$ (cosine) between the entity's orientation axis and the direction toward `target` (another entity or a frame direction). The `code` tool discretizes it into categorical labels (facing toward / away / perpendicular). Computed from `entity_orientation_axis` and frame-transformed displacement.
  - Example: `facing_alignment(painting, north, frame)`.
- **containment_relation(entity_inner, entity_outer)**: Returns a Boolean and a containment ratio (fraction of inner points inside outer's bounding volume). Used for inside/within queries. Computed from `entity_pointset(entity_inner)` and `entity_bbox3d(entity_outer)`.
  - Example: `containment_relation(book, shelf_compartment)`.
- **between_ratio(entity_x, entity_a, entity_b, frame)**: Returns a scalar in $[0, 1]$ indicating how close `entity_x` is to the midpoint of the segment between `entity_a` and `entity_b`, along with a perpendicular offset. Used for between-type queries. Computed from `entity_centroid` of the three entities. Example: `between_ratio(chair, sofa, table, frame)`.
- **count_metric(candidate_entity_set)**: Returns the number of grounded instances in a candidate entity set. Used for "how many" queries. Computed from the cardinality of the grounded set after Entity-level disambiguation. Example: `count_metric(candidate_entity_set={chair})`.

- **camera_translation(view_a, view_b, frame)**: Returns the 3D translation vector of the camera from `view_a` to `view_b` in the reference frame. Used for "in which direction did I move" queries. Computed from `camera_pose` of both views.
  - Example: `camera_translation(view_1, view_2, frame)`.
- **camera_rotation(view_a, view_b, frame)**: Returns the relative rotation of the camera from `view_a` to `view_b`, as a rotation matrix decoded by the `code` tool into rotation vector or Euler angles. Used for "how did I turn" queries. Computed from `camera_pose` of both views.
  - Example: `camera_rotation(view_1, view_2, frame)`.

- **region_object_query(region_spec, candidate_entity_set, frame)**: Returns the subset of entities in `candidate_entity_set` whose centroids fall inside a 3D region defined relative to an anchor entity or camera under the frame (e.g., "left of `entity_a`", "in front of `view_3`"). Used for "what is to the left/right/front" queries pervasive in multi-view spatial reasoning. Computed from `entity_centroid` of each candidate and a region-membership test.
  - Example: `region_object_query(left_of(chair), candidate_entity_set={table, bin, sofa}, frame)`.

Table 14: **Prompt Used for Per-Question Primitive Decomposition.** This prompt is applied to each sampled question independently to decompose it into candidate spatial primitives across the four FESM layers. Here, {question} is the placeholder that will be replaced.

[CORE MISSION]
You are an expert spatial reasoning analyst. Your sole mission is to analyze a user's spatial reasoning question and decompose it into the minimal set of information that a system must determine before the question can be solved deterministically. You should not attempt to solve the question. You should only identify what must be known. Ask yourself: *"What are the irreducible pieces of information that any solver, regardless of its implementation, would need to commit to before computing the answer?"*
[OUTPUT FORMAT]
Your response **MUST** be a single, valid JSON object:

```
{
  "reasoning": "A brief, step-by-step logical deduction that explains what
               information the question depends on and why.",
  "primitives": {
    "Frame": [{"name": "...", "description": "...", "rationale": "..."}],
    "Entity": [{"name": "...", "description": "...", "rationale": "..."}],
    "State": [{"name": "...", "description": "...", "rationale": "..."}],
    "Metric": [{"name": "...", "description": "...", "rationale": "..."}]
  }
}
```

[DECOMPOSITION FRAMEWORK]
Every spatial reasoning question implicitly requires the solver to commit to four categories of information. You must identify what the question demands in each category. If a category is not required by the question, return an empty list for that category.

- **Frame**: What reference frame must be established to interpret the question? A reference frame is the coordinate system under which directional words such as *left, right, front, behind, north, south* are grounded. Identify what defines the reference frame. For example, is it defined by a camera viewpoint, by an object's orientation, by a direction between two objects, or by none (if the question is frame-agnostic such as counting)?

- **Entity**: Which concrete scene entities must the solver ground in order to answer the question? An entity is any object, region, or camera mentioned or implied by the question. Identify the anchor entity (the reference), the target entity (what is being asked about), and any disambiguating constraints (e.g., a region filter such as *"on the wall"*, a view binding such as *"in image 3"*, or a cardinality expectation such as *"the sofa"* implying exactly one).

- **State**: What geometric information about the entities must be acquired? Identify the minimum geometric state each entity requires. For example, does the question depend only on position (centroid), on full shape (point set), on size (bounding box or extent), or on orientation (pose or a single axis)?

- **Metric**: What measurable quantity must ultimately be computed to answer the question? Identify the top-level relational computation. For example, is it a displacement vector, a distance, a vertical offset, an angular difference, a facing alignment, a containment test, a count, a camera motion, or a region-membership query?

[RULES]
- **Do not invent computational procedures**. You are identifying requirements, not solution steps. A good primitive is a piece of information the solver must know. A bad primitive is a step the solver must execute.
- **Use concise, canonical names**. Name each primitive with a short, descriptive identifier in `snake_case` (e.g., `anchor_entity`, `entity_orientation_axis`, `planar_displacement`). Avoid verbose or question-specific names.
- **One primitive per distinct requirement**. If the question requires two different entities to be grounded, list both. If it requires two different geometric states (e.g., position of one entity and orientation of another), list both.
- **Rationale must reference the question**. Each primitive's `rationale` must point to the specific phrase or semantic element in the question that demands it.
- **No overlap across categories**. A reference frame definition belongs in Frame, not in Entity. A target object belongs in Entity, not in Metric. Keep each primitive in the category where its role is defined.

[QUESTION]
{question}

Now, please analyze the above question and provide your response in the specified JSON format.

Table 15: **Prompt Used for Cross-Question Schema Aggregation.** After per-question decomposition (Table 14), this prompt aggregates the decomposition results across all sampled questions to induce the initial FESM schema. Here, {per_question_decompositions} is the placeholder for the concatenated JSON outputs from Stage A.

[CORE MISSION]
You are an expert spatial reasoning analyst. You are given the per-question primitive decompositions of a large set of sampled spatial reasoning questions. Your sole mission is to aggregate these decompositions into a compact, non-redundant, typed schema that covers the spatial reasoning requirements observed across the questions. You should not add primitives that are not supported by the decompositions. You should not retain primitives that appear only as surface-level paraphrases of other primitives. Ask yourself: *"What is the minimal set of typed spatial primitives under which every sampled question can be faithfully re-expressed?"*
[OUTPUT FORMAT]
Your response **MUST** be a single, valid JSON object:

```
{
  "reasoning": "A brief summary of how the per-question decompositions were
               grouped, merged, and organized into the final schema.",
  "schema": {
    "Frame": [
      {
        "name": "...",
        "description": "...",
        "valid_values": ["..."],
        "supporting_question_ids": [...],
        "coverage_count": N
      }
    ],
    "Entity": [{...}],
    "State": [{...}],
    "Metric": [{...}]
  },
  "discarded": [{"name": "...", "reason": "merged into X" | "redundant with Y" | ...}]
}
```

[AGGREGATION PROCEDURE]
- **Group by semantic equivalence**: Primitives that appear under different names across questions but describe the same underlying requirement must be merged. For example, `anchor_orientation`, `anchor_facing_axis`, and `reference_front_direction` all refer to the same concept and should be unified under one canonical primitive.
- **Unify parameterization**: If two primitives differ only in their parameters or argument values (e.g., `centroid_of_chair` vs. `centroid_of_table`), merge them into a single parameterized primitive (e.g., `entity_centroid(entity)`).
- **Keep distinct what is truly distinct**: Two primitives that sometimes co-occur but can also appear independently across different questions must remain separate. Do not over-merge.
- **Enumerate valid values where applicable**: If a primitive has a small closed set of values observed across the decompositions (e.g., `frame_type` taking values `camera_based`, `object_based`, `direction_based`), list them explicitly in `valid_values`.
- **Assign each primitive to exactly one FESM category**: Frame, Entity, State, or Metric. If a primitive appears to span two categories in some decompositions, inspect its core role and place it where it is defined.
- **Record coverage**: For each primitive in the final schema, record the list of question IDs whose decomposition contributed to it, and the total count. This serves as evidence that the primitive is grounded in observed questions rather than speculated.

[CONSOLIDATION RULES]
- **Minimum support**: A primitive is included in the final schema only if it is supported by at least τ_{support} questions. Primitives with fewer supports are listed in the `discarded` field with reason ``insufficient support''.
- **No primitive without a computation path**: Each primitive in the final schema must be acquirable from the fixed geometric tool set available to downstream modules (3D reconstruction, detection, segmentation, object pose estimation, scale estimation, OCR, motion analysis, and code execution). If a candidate primitive cannot be grounded in these tools, list it in the `discarded` field with reason ``not tool-computable''.
- **Prefer coarse over fine when sufficient**: When two primitives express the same requirement at different granularities (e.g., `entity_centroid` and `entity_pointset`) and the coarser one is sufficient for most observed questions, retain both but mark the coarser one as the default.
- **Canonical naming**: All final primitives must use short, descriptive `snake_case` identifiers. Avoid verbose names or names that embed specific entity categories.

[PER-QUESTION DECOMPOSITIONS]
{per_question_decompositions}

Now, please aggregate the above decompositions and provide the induced FESM schema in the specified JSON.

Table 16: **Prompt Used for Retrieving FESM Requirements Library.** Here, {library} is the predefined FESM library, {examples} is the in-context examples, and {question} is the placeholder that will be replaced by the user's query with input images.

[CORE MISSION]
You are an expert spatial reasoning requirement analyst. Your sole mission is to analyze the user's question together with the input images, and retrieve the minimal but sufficient set of required items from the predefined **Frame–Entity–State–Metric (FESM) library**. Your goal is **NOT** to answer the question, **NOT** to perform geometric computation, **NOT** to decide tool calls, and **NOT** to verbalize the final answer. Your task is only to determine: *"What must the system know, ground, and acquire before this query can be solved deterministically?"*

[OUTPUT FORMAT]
Your response **MUST** be a single, valid JSON object:

```
{"reasoning": "A brief, step-by-step logical deduction explaining why the selected
items are necessary and why no extra items are needed.",
  "requirements": {
    "Frame": [
      "Exact library entries required to define the reference frame"
    ],
    "Entity": [
      "Exact library entries required to ground entities"
    ],
    "State": [
      "Exact library entries required to acquire geometric states"
    ],
    "Metric": [
      "Exact library entries required to compute the final relation"
    ]}}
```

[RETRIEVAL PRINCIPLE]
1. Retrieve Requirements in Fixed Order
You MUST retrieve requirements strictly in the following order: **Frame** → **Entity** → **State** → **Metric**.
- **Priority 1: Frame.** First determine under which reference frame the query should be interpreted.
- **Priority 2: Entity.** Then determine which concrete scene entities the query refers to.
- **Priority 3: State.** Then determine which geometric states must be acquired for those entities.
- **Priority 4: Metric.** Finally determine which measurable quantity should be computed to solve the query.

2. Retrieve Frame Requirements
To retrieve **Frame**, you MUST identify the element that gives final authority to spatial directions such as "front", "behind", "left", "right", or cardinal directions such as "north". You MUST retrieve only the entries necessary for the chosen frame type. Do not mix frame types unless the query explicitly requires it.

3. Retrieve Entity Requirements
To retrieve **Entity**, you MUST identify which scene entities are actually involved in the query.
- **A. Relation Anchor**: Retrieve the entity that serves as the spatial reference in the queried relation.
- **B. Target Entity**: Retrieve the entity whose position, orientation, or relation is being queried.
- **C. Candidate Set**: Retrieve a candidate set only when the query asks to search over multiple possible entities.
- **D. Disambiguation Constraints**: Retrieve region filters, attribute filters, or contextual constraints only when they are necessary to uniquely ground the intended entity.
- Do not assume that the frame anchor and the relation anchor are always the same.

4. Retrieve State Requirements
To retrieve **State**, you MUST identify which geometric states must be acquired before any computation.
- **A. Prefer Sufficient Primitive States**: Retrieve the simplest state that is sufficient for downstream computation.
- **B. Coarse States First**: Prefer coarse states such as centroid or 3D box when they are enough.
- **C. Fine States on Demand**: Retrieve pose, front axis, normal axis, point set, or camera pose only when the query explicitly depends on orientation, dense geometry, or viewpoint.
- Do not retrieve finer states if a simpler state is already sufficient.

5. Retrieve Metric Requirements
You MUST identify the final measurable quantity that directly solves the query once the required states are available.

[LIBRARY]
{library}

[EXAMPLES]
{examples}

[QUESTION]
{question}

Now, please analyze the above question and provide your response in the specified JSON format.

Table 17: **Prompt Used for Requirement-Guided Tool Orchestration.** Here, {api_documents} is the documentation of available APIs, {requirements} is the retrieved requirement set, and {history} contains the user question, current workspace, previous plans, and execution results.

[CORE MISSION]
You are an expert spatial reasoning executor. Your mission is to acquire the missing values required by the pre-retrieved **Frame–Entity–State–Metric (FESM)** requirements. You are **not** allowed to freely redesign the solving procedure. You must use the requirement set as the only target, inspect the current workspace, and generate the **immediate next tool calls** needed to fill the missing variables.

[AVAILABLE APIS]
{api_documents}

[REQUIREMENT SET]
{requirements}

[EXECUTION POLICY]
1. Follow the Requirement Set Strictly
- The requirement set already specifies **what must be known**. You must not invent new goals beyond it.
- Your job is only to determine **how to acquire the still-missing values** needed by the requirement set.
- Always reuse variables already stored in the workspace before calling new tools.

2. Acquire Required Values: Based on the user's question and the pre-retrieved **Frame–Entity–State–Metric** requirement set, plan the necessary tool calls to acquire all missing values needed for deterministic computation. This process involves four dependency-coupled goals:
- **A. Frame Requirements**: First establish the reference frame.
  - If the frame depends on a **camera**, call `reconstruct` to obtain reconstruction context and camera extrinsics.
  - If the frame depends on an **object's orientation**, first ground that object, then call `predict_obj_pose`.
  - If the frame depends on a **direction between two objects**, first ground both objects, then acquire their 3D representative points (e.g., centroids) using APIs such as `project_box_to_3d_points`.
- **B. Entity Requirements**: Then ground the queried entities.
  - Use `detect` to localize the required anchor entity, target entity, or candidate entity set in the image.
  - If the query contains region, attribute, or text constraints, use them only for disambiguation.
  - When a grounded entity is needed in 3D, call `project_box_to_3d_points` on its detected box to obtain its corresponding 3D points in the world frame; use `ocr` when text is part of the grounding clue.
- **C. State Requirements**: Then acquire the geometric states required by the retrieved requirements.
  - For orientation-based states such as object pose or local axes, call `predict_obj_pose`.
  - For metric-distance queries, call `estimate_scale` before performing meter-level computation.
  - For subtle view-change or small camera-motion queries, call `analyze_motion`.
  - Once all required state variables are available in the workspace, call `code` to compute the required metric.
- **D. Metric Requirements**: Finally prepare the inputs needed for the required metric.
  - Only when all prerequisite variables are available, call `code` to compute the metric deterministically.

4. Use Tools to Fill Variables, Not to Guess Answers
- Tool calls must be used to obtain concrete variables required by the requirement set.
- Do not infer the final answer during this stage.
- Do not verbalize, judge, or conclude the queried relation before the required values are available.

[OUTPUT FORMAT]
Your response **MUST** be a single, valid JSON object:

```
{
  "analysis": "Briefly explain which required variables are still missing in the
               workspace, and why the proposed tool calls are the immediate next step.",
  "missing_requirements": [
    "List the unresolved requirement items that motivate the next action"
  ],
  "tool_calls": [
    {
      "api": "API name",
      "args": {...},
      "output_variable": "A unique variable name to store the result in workspace"
    }
  ]
}
```

[HISTORY]
Here is the history so far: {history}

Please examine the requirement set and the current workspace, then generate the next-step response. Your plan must include **only the immediate next action**, not the full solution.

Table 18: **Prompt Used for Coder.** Here, {question} is the original user query, {requirements} is the retrieved Frame–Entity–State–Metric requirement set, {workspace_docs} documents the currently available variables in the workspace, and {knowledge} contains the fixed geometric formulas and rules relevant to the required metric.

[CORE MISSION]
You are an expert geometric computation programmer. Your task is to write a single Python function that deterministically computes the result required by the retrieved **Frame–Entity–State–Metric** requirements, using only the variables already available in the workspace.

[QUESTION]
This is the original high-level query that provides the task context.
{question}

[RETRIEVED REQUIREMENTS]
The query has already been formalized into a **Frame–Entity–State–Metric** requirement set. You must follow this requirement set strictly.
{requirements}

[WORKSPACE VARIABLES]
These are the variables that have already been acquired by previous tool calls. Your code must use these variables as the only computational inputs.
{workspace_docs}

[FIXED GEOMETRIC KNOWLEDGE]
These formulas and rules are valid and can be used directly in your implementation.
{knowledge}

[AVAILABLE LIBRARIES]
You may use `numpy`, `torch`, `scipy`, `math`, and other standard Python libraries.

[CODING INSTRUCTIONS]
1. Compute Only from Available Variables
- You must compute the result only from the variables documented in the workspace.
- Do not assume any missing variable, and do not invent any new input.
- If a required quantity must be derived, derive it from the provided variables.

2. Respect the Retrieved Requirements
- Follow the **Frame** requirements when interpreting directions such as left/right/north/south.
- Use the **Entity** requirements to ensure that computation is performed on the correct grounded entities.
- Use the **State** requirements to identify which geometric inputs are relevant.
- Use the **Metric** requirements to determine what must be computed as the final quantity.

3. Perform Deterministic Computation
- Your function must implement explicit geometric computation, not heuristic guessing.
- If coordinate transformation is required, transform the relevant variables into the required reference frame before judging the final relation.
- If the final result is categorical, compute the necessary geometric quantity first, then map it to the final label or option.

4. Handle Multiple-Choice Questions Carefully
- If the question is multiple-choice, your code must evaluate the given options systematically.
- Restrict the decision process to the provided options only.
- Do not introduce new answer candidates beyond the given choices.

5. Output Only Code
Your final output **MUST** be a single Python function in the following format:

```
``` python
def execute(func_signature):
    # Your code here
    ...
    return serializable_value
```
```

The return value must be serializable. Do not include any explanation, markdown, or extra text outside the function.

Table 19: Prompt for Verifier Candidate Evaluation

[CORE MISSION]
You are an expert verifier evaluating the quality of a program that solves a 3D spatial reasoning problem with tools and primitives. The candidate is a structured execution outcome consisting of: a predicted answer and corresponding execution trace. Your job is to: decide whether this candidate should be accepted or rejected; assign a quality score from 1.0 to 10.0; and, if rejected, return structured feedback that explains whether the failure comes from incomplete use of the existing primitive schema or from a primitive schema gap.

[Question]
{question}

[Input Images]
{Images}

[Program to Evaluate]
{`program_code`}

[Execution Trace]
– **Execution Status: {success/failure/partial}
– **Final Answer: {answer_text}
– **Used APIs / Tools: {used_tools}
– **Execution Error: {execution_error if present}

[EVALUATION FRAMEWORK]
1. Query–Answer Consistency
Check whether the predicted answer actually addresses the user's question.
– Does the answer correspond to the queried relation, object, direction, or quantity?
– Is it semantically compatible with the query?
– If the question is multiple–choice, is the returned answer consistent with the provided options?
2. Visual Evidence Verification
Use the input images as grounding evidence throughout your evaluation. Check whether:
– the answer is visually plausible,
– the identified entities match the visible scene,
– the relevant spatial relation is compatible with the observed 3D structure,
– the intermediate results in the trace are visually consistent with the scene.
3. FESM Requirement Coverage
The retrieved primitives set defines what the system committed to knowing before solving the query. You must check whether the execution trace sufficiently fulfills these commitments. Specifically verify:
– Frame: Was the reference frame fully grounded? If directional interpretation is required, is the frame anchor, forward axis, and vertical axis sufficiently established?
– Entity: Were the required entities correctly grounded and disambiguated? Were view constraints, region filters, or expected–count constraints respected?
– State: Were the required geometric states acquired at the appropriate granularity? Was a coarse state used where sufficient, and was a finer state retrieved when required?
– Metric: Was the final metric actually computable from the available instantiated states?
Pay close attention to any primitive marked as partial, failed, unresolved, or only weakly supported.
4. Failure Attribution
If the candidate should be rejected, you must identify which of the following two cases applies:
A. Missing-slot feedback
Use this when the current schema is already expressive enough, but the retrieved or instantiated primitive set is incomplete.
B. Primitive-gap feedback
Use this only when the current schema itself lacks the representational capacity needed by the query. Typical signs include:
– the query requires a type of frame, entity constraint, state, or metric that cannot be expressed by the current primitives,
– the trace fails not because a known primitive was omitted, but because no existing primitive could represent the need,
– repeated partial workarounds are visible, but the missing concept is not expressible within the current schema.

[SCORING CRITERIA (1.0 - 10.0 Scale)]

1. Requirement Satisfaction
– How completely does the execution trace fulfill the retrieved FESM requirements?
– Are all necessary primitives properly instantiated?
2. Geometric Soundness
– Is the reasoning grounded in valid geometric evidence?
– Does the final computation follow from the instantiated states and metric?
3. Visual and Semantic Correctness
– Is the answer consistent with both the visual scene and the query semantics?
– Are grounded entities and relations plausible given the images?
4. Trace Reliability
– Does the trace show a clear and valid requirement–to–computation path?
– Are intermediate variables and tool outputs coherent and usable?
5. Diagnostic Confidence
– Can the success or failure be clearly attributed?
– If rejected, is the failure type identifiable as missing–slot or primitive–gap?

[OUTPUT FORMAT]
You **MUST** provide your response in exactly this format:

\<verification_decision>accept|reject</verification_decision>
<quality_score>NUMBER</quality_score>
<reasoning>
[Detailed explanation covering:
– whether the answer is supported by the query, images, and execution trace,
– whether the retrieved FESM requirements are sufficiently fulfilled,
– whether the final computation is geometrically well–supported,
– what specific strengths or failures are visible in the trace,
– and why the candidate should be accepted or rejected.]
</reasoning>
<feedback_type>none|missing_slot|primitive_gap</feedback_type>
<feedback>
[If verification_decision=reject, provide a concise but precise structured feedback message.
Examples:
– ”frame grounding incomplete; target entity grounded but anchor orientation missing”
– ”metric requires planar projection before categorical left/right judgment”
– ”current schema lacks event–defined orientation primitive”
If verification_decision=accept, output exactly: none]
</feedback>