SpatialClaw: Rethinking Action Interface for Agentic Spatial Reasoning

Seokju Cho[1,2*] **Ryo Hachiuma**[1] **Abhishek Badki**[1] **Hang Su**[1]
Byung-Kwan Lee[1] **Chan Hee Song**[1] **Sifei Liu**[1] **Subhashree Radhakrishnan**[1]
Seungryong Kim[2] **Yu-Chiang Frank Wang**[1] **Min-Hung Chen**[1]

[1]NVIDIA [2]KAIST AI

`github.com/NVlabs/SpatialClaw` `spatialclaw.github.io`

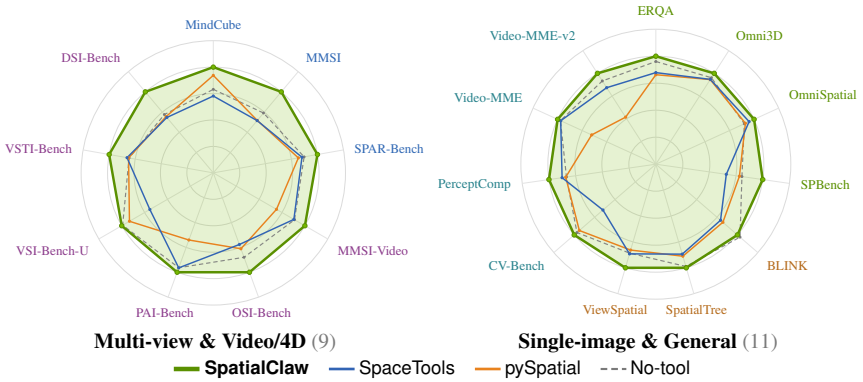


Figure 1: **SpatialClaw improves spatial reasoning across the board.** Per-benchmark accuracy on **20 spatial reasoning benchmarks** (Gemma 4-31B backbone), split into two panels by task category. Baselines are SpaceTools-Toolshed [6], pySpatial [28], and a no-tool backbone.

Abstract

Spatial reasoning, the ability to determine where objects are, how they relate, and how they move in 3D, remains a fundamental challenge for vision-language models (VLMs). Tool-augmented agents attempt to address this by augmenting VLMs with specialist perception modules, yet their effectiveness is bounded by the *action interface* through which those tools are invoked. In this work, we study how the design of this interface shapes the agent's capacity for open-ended spatial reasoning. Existing spatial agents either employ single-pass code execution, which commits to a full analysis strategy before any intermediate result is observed, or rely on a structured tool-call interface that often offers less flexibility for freely composing operations or tailoring the analysis to each task. Both designs offer limited flexibility for open-ended, complex 3D/4D spatial reasoning. We therefore propose **SpatialClaw**, a training-free framework for spatial reasoning that adopts code as the action interface. SpatialClaw maintains a stateful Python kernel preloaded with input frames and a suite of perception and geometry primitives, letting a VLM-backed agent write one executable cell per step conditioned on all prior outputs, enabling the agent to flexibly compose and manipulate perception results and adapt its analysis to both intermediate text and visual observations and the demands of each problem. Evaluated across **20 spatial reasoning benchmarks** spanning a broad range of static and dynamic 3D/4D spatial reasoning tasks, SpatialClaw achieves 59.9% average accuracy, outperforming the recent spatial agent by **+11.2 points**, with consistent gains across eight VLM backbones from two model families *without any benchmark- or model-specific adaptation*.

*Work done during an internship at NVIDIA.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

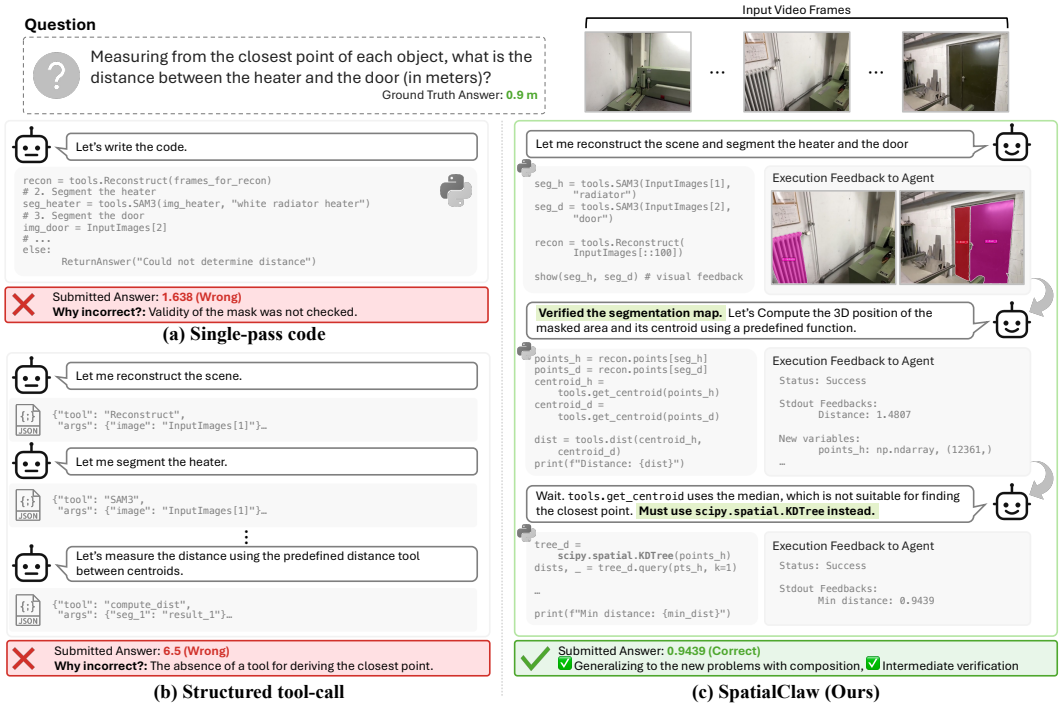


Figure 2: **SpatialClaw studies code as the action interface for spatial reasoning.** Three action interfaces on the same question. (a) A single-pass program chooses a complete computation before seeing its intermediate outputs. (b) A structured tool interface exposes common operations through structured commands (*e.g.*, JSON, XML). (c) SpatialClaw writes Python in a persistent kernel, renders intermediate evidence, and revises the measurement before answering.

1 Introduction

Spatial reasoning, the ability to determine where objects are, how they move, and how they relate in three dimensions, is foundational to visual perception. Questions such as *"Is the car moving toward the camera?"*, *"Which object is closest to the table?"*, and *"Did the person turn left or right?"* are effortless for humans, yet remain challenging for vision-language models (VLMs), including state-of-the-art models [49, 5, 31, 14, 20]. Answering them requires composing multiple types of evidence, including depth, camera pose, and temporal correspondence, into a coherent geometric argument, and current VLMs struggle to perform this structured analysis reliably from pixels alone [4, 7, 8].

A natural response is to augment VLMs with specialist perception tools [15, 38, 36, 27, 6, 16], such as detectors [2], segmenters [34, 3], and depth or pose estimators [45], that produce perceptual outputs that VLMs would otherwise have to approximate from visual appearance alone. The capability of a tool-augmented agent, however, depends not only on which tools are available, but also on the *action interface* through which those tools are invoked. The action interface specifies how tools are invoked and their outputs represented, which intermediate states are observable between steps, and whether the agent can revise its reasoning in light of those observations before committing to an answer.

Prior tool-augmented spatial agents have primarily relied on one of two action interfaces. In *single-pass code execution*, the agent writes a complete Python program and runs it once [28, 29] (Fig. 2a), enabling flexible tool invocation but requiring a complete analysis strategy to be committed before any intermediate result has been observed. In *structured tool-calls*, the agent selects from a list of named tools, fills typed arguments, and consumes typed return values [51, 6, 35], offering limited support for composing and synthesizing tool outputs through external scientific libraries such as `numpy` or `scipy` to perform task-specific computations that emerge only at test time (Fig. 2b).

In practice, neither design readily supports the open-ended, compositional reasoning that complex 3D/4D spatial reasoning tasks often demand. We therefore argue that the appropriate action interface for spatial reasoning should treat code not as a one-shot program or a dispatch interface for pre-

registered tools, but as an *orchestration space* in which the agent sequences, composes, and diagnoses the perception tools it has access to. In light of this, we propose **SpatialClaw**, a training-free framework that instantiates this principle as code as the action interface (Fig. 2c).

In SpatialClaw, perception tools are Python callables and their outputs, including masks, depth maps, camera geometry, and trajectories, are ordinary Python variables; the kernel preserves this state independently across turns, so any object produced at one step remains available for composition, inspection, and revision at all subsequent steps. While structured tool-calls and natural language reasoning can partially approximate this behavior, code generation provides a more expressive interface for adapting perception to the task, especially in spatial reasoning, where the required computations often cannot be anticipated by a fixed API. A new spatial analysis is not a new API entry, but a new composition of perception tools (e.g., segmentation [3] or reconstruction [25]) and numerical primitives (e.g., `numpy` [17], `scipy` [42]) assembled across steps in response to intermediate evidence. SpatialClaw pre-loads a persistent Python kernel with input frames, perception tools, and scientific libraries, and coordinates a VLM-backed agent through an iterative loop of planning, code execution, and feedback assembly, guided by a unified system prompt that encodes general principles of spatial reasoning rather than task-specific instructions.

We evaluate SpatialClaw on 20 spatial reasoning benchmarks spanning metric distance, camera and object motion, multi-view geometry, temporal reasoning, and spatial planning, to name a few. SpatialClaw achieves 59.9% average accuracy across all 20 benchmarks, outperforming the recent spatial agent [6] by **+11.2 points** (Fig. 1). Gains are largest on dynamic 4D video reasoning and multi-view inference, precisely the categories that demand chained geometric computation across frames and viewpoints, where no pre-specified tool call captures the required composition. Strikingly, these gains generalize out of the box: SpatialClaw delivers consistent improvements across the large majority of benchmarks and backbone models tested, spanning two distinct model families, Qwen [31–33] and Gemma4 [14], with parameters ranging from 27B to 397B, *without any modification* to the system prompt, tool set, or benchmark-specific engineering, confirming that the expressive action interface itself drives the gains rather than model-specific tuning. We further present a controlled comparison of three action interfaces and ablation studies, demonstrating that our interface achieves the strongest generalization through flexible composition of perception primitives and that the gains persist even when all pre-defined utility wrappers are removed.

Our contributions are as follows:

- **Rethinking the action interface for spatial reasoning agents.** We argue that the design of the action interface is a critical factor in agent capability in spatial reasoning, and introduce SpatialClaw, a training-free agent that instantiates this principle by replacing single-pass code execution and structured tool-call interfaces with a persistent, multi-turn Python kernel, turning each step into an opportunity to compose perception outputs with numerical primitives, inspect intermediate state, and revise the analysis.
- **Comprehensive evaluation that validates action interface design.** We establish an extensive evaluation spanning single-image spatial reasoning, multi-view spatial reasoning, general spatial reasoning, video spatial & 4D reasoning, and general video understanding, on which SpatialClaw achieves consistent improvement over baselines across most benchmarks and transfers across backbone models without any benchmark- or model-specific adaptation, providing a comprehensive empirical comparison of spatial agents.

2 Action Interfaces for Spatial Reasoning Agents

We characterize tool-augmented spatial agents by their *action interface*, that is, the medium through which they acquire, inspect, and transform visual evidence. When no interface is used, the VLM performs spatial reasoning by generating a chain-of-thought rationale in natural language directly over the input images [49, 5, 31, 14], without invoking any external computation or acquiring intermediate evidence. Beyond this tool-free case, prior spatial agents have primarily relied on one of two action interfaces: single-pass code execution, or predefined API calls through structured tool-calls.

Single-pass code. The agent writes a complete Python program, runs it once, and uses it to call perception modules and process their outputs [28, 29]. The program can express task-specific computations, but it must commit to a complete strategy before observing any intermediate mask,

depth map, plot, or runtime error. Retry mechanisms may repair a syntax or runtime failure, but intermediate visual evidence does not participate in the reasoning loop.

Structured tool-calls. The agent invokes perception modules by name, passing typed inputs and receiving typed outputs through predefined structured commands [51, 6, 35], e.g., JSON or XML. While this exposes a clean interface for invoking perception, compositions that can only be determined at test time, such as chaining a depth estimate with a segmentation mask in a way specific to the question, are difficult to express within the predefined command schema.

Motivation. These considerations motivate a third interface centered on code as a medium for flexible iterative composition based on intermediate evidence. Building on the rapid advances in code generation capabilities of modern LLMs, SpatialClaw lets the agent write one Python cell per step and execute it in a persistent kernel, conditioning each subsequent step on the resulting text, variables, errors, and visualizations; that is, we interpret code as the action interface. Code generation provides a more general mechanism for adapting to the task than either approach: computations such as finding the nearest object via `scipy.spatial.KDTree` or estimating a surface plane via RANSAC [10] emerge naturally from the problem at hand, without needing to be anticipated by a predefined interface. Spatial results such as depth maps, camera geometry, and temporal trajectories persist as ordinary Python variables across steps, remaining available for inspection, composition, and correction throughout execution. Fig. 2 provides a comparison of these three interfaces.

3 SpatialClaw

SpatialClaw realizes this code as the action interface through a persistent Python workspace for spatial reasoning. Rather than exposing a larger menu of spatial tools, it lets the agent turn visual evidence into inspectable spatial analyses that can be composed and revised over multiple steps. For each example, SpatialClaw creates a persistent Python kernel pre-loaded with the input frames and a set of primitives spanning perception modules (e.g., depth estimation, segmentation) and scientific libraries (e.g., NumPy [17], SciPy [42], Matplotlib [19]), and asks a VLM-backed agent to write one executable code cell at a time. Each cell can create masks, reconstructions, plots, or numeric summaries, and the resulting program state remains available to later cells. The next action is conditioned on text output, variable summaries, and rendered intermediate images from previous steps. The remainder of this section formalizes these two components: the persistent kernel workspace that maintains shared program state across execution steps (§3.1), and the outer agentic loop that coordinates planning, code generation, feedback assembly, and answer submission (§3.2).

3.1 Persistent Kernel Workspace

The workspace is initialized once for an example and discarded after the system terminates. It keeps all intermediate results as ordinary Python variables, so any object produced during execution, including segmentation masks, depth maps, numeric arrays, and rendered plots, remains accessible to subsequent code cells. As a result, the agent does not need to choose the complete analysis in advance. It can start with a coarse computation, inspect the result, and refine the analysis against the observed evidence. The kernel exposes six public entry points.

1. `InputImages` contains the sampled frames or images.
2. `Metadata` contains frame rate, duration, and frame indices for video inputs, enabling the agent to reason about temporal structure when questions reference specific timestamps.
3. `tools` exposes evidence-producing perception and geometry primitives. `Reconstruct` wraps Depth Anything 3 [25] and returns per-frame depth, camera intrinsics, camera extrinsics, and dense point maps. `SAM3` [3] produces image or video masks from text, point, or box prompts. The `tools` module also includes lightweight utilities for mask operations, geometric computations, and visualizations. These utility functions simplify common operations such as mask manipulation and geometric computation, but the agent is not limited to what they provide; any operation can also be implemented directly over arrays or through the scientific libraries (`NumPy`, `SciPy`, `Matplotlib`) available in the execution environment. Full descriptions of all tools and utilities are provided in Appendix G.

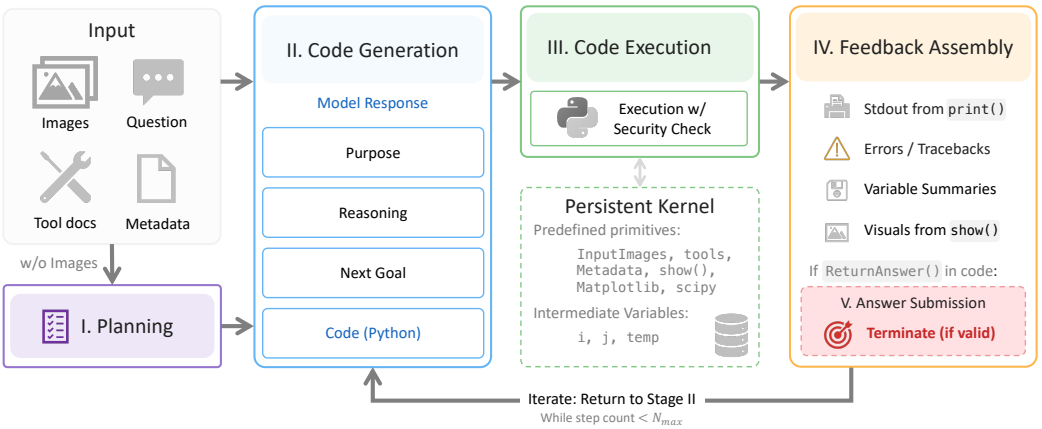


Figure 3: **Agentic loop for iterative code execution.** SpatialClaw wraps a persistent kernel in a five-stage loop. A planner receives the question and tool documentation but not the images, and produces an analysis plan. The main agent generates a Python cell executed in the persistent kernel. Feedback comprising stdout, variable summaries, and images registered via `show()` is appended to the model context. The loop continues until the agent submits an answer with `ReturnAnswer()` or the step count has reached the predefined maximum N_{max}.

4. `show(...)` registers an image to be embedded directly into the agent's context at the next step, allowing the agent to visually inspect intermediate results such as masks, depth maps, or annotated frames before deciding what to do next. Figures rendered via `matplotlib` (`plt.show()`) are captured in the same manner and included in the agent's next observation.
5. `vlm` dispatches queries to a separate VLM session, returning the response as plain text without embedding images into the main agent's context. `vlm.locate(...)` performs visual grounding, returning bounding boxes. `vlm.ask_with_thinking(...)` handles questions that fall outside the perception toolset, such as those requiring commonsense reasoning or artistic style recognition. Further details on the prompts used for each VLM session are given in Appendix F.
6. `ReturnAnswer(...)` submits a candidate answer.

Together, these entry points form a self-contained computational environment: the agent reads observations through `InputImages` and `Metadata`, builds up spatial evidence via `tools`, inspects intermediate results through `show(...)`, queries a separate VLM session when needed via `vlm`, and commits a final answer through `ReturnAnswer(...)`. How the agent sequences these actions across multiple steps is governed by the outer loop described next.

3.2 Spatial Reasoning Loop: Code, Inspect, and Revise

The persistent kernel provides a runtime environment for the action interface, but does not by itself prescribe when to plan, when to act on the observed evidence, and when to commit to an answer. We therefore wrap the kernel in a five-stage outer loop (Fig. 3) covering planning, code generation, code execution, feedback assembly, and answer submission. The loop follows standard agent control structures and serves as a stable scaffold under which the construct, inspect, and revise pattern operates consistently across examples. The loop prescribes the order and termination conditions of each stage, but not how the agent should reason spatially within them; that reasoning discipline is encoded in the system prompt. Details of the agent system are provided in §E.

Disciplined reasoning via system prompt. SpatialClaw imposes structure on the open-ended Python action space through a unified system prompt, without any benchmark-specific engineering. The prompt defines both the runtime objects available in the kernel and the verification discipline the agent is required to follow. The prompt encourages the agent to treat spatial conclusions as claims that should be cross-checked against multiple evidence sources where possible. For instance, the agent is guided to resolve the relevant frame of reference, prefer metric computation over pixel-level impressions for geometric questions, and visually inspect tool outputs such as masks and annotated objects. It is also encouraged to sanity-check numerical magnitudes and consider any apparent

disagreements between visualizations and computed quantities before invoking `ReturnAnswer()`. Because the prompt encodes general principles rather than few-shot examples or task-specific templates, the same configuration is applied across all benchmarks and backbone models without modification; more prompt details are given in §F. With this discipline in place, the loop proceeds through five stages as follows:

Stage I: Planning. The planner runs in a separate LLM session, isolated from the main agent's execution context, with its own dedicated system prompt, and is invoked once per sample prior to execution. Unlike the main agent's prompt, which focuses on general spatial reasoning discipline, the planner's prompt forbids executable code and pre-conclusion phrases, and instead instructs the planner to outline the analysis steps and the evidence to be acquired. It receives the question, metadata, and tool documentation, but not the input frames for efficiency. The resulting plan is appended to the main agent's system prompt before execution begins, providing the agent with a structured plan to follow.

Stage II: Code generation. The main VLM-backed agent receives the question, the plan, the execution trajectory, and any images shown in previous steps. At each step, it produces a structured response comprising *purpose*, *reasoning*, *next goal*, and *code* fields, formatted in markdown. The *code* field is a Python cell that implements the next analysis action, which can call perception primitives, transform their outputs, render intermediate evidence, or submit an answer via `ReturnAnswer()`.

Stage III: Code execution. Before execution, a static checker parses the code's abstract syntax tree (AST) to reject disallowed modules and unsafe builtins without running the code; on failure, the framework injects a concise traceback into the context and prompts the agent to revise the code. The validated cell is then executed in the persistent kernel.

Stage IV: Feedback assembly. The next model context receives the standard output produced by the executed cell (e.g., values printed via `print()`), concise tracebacks for any failed execution, summaries of newly created variables including their type, length, and size, and any images registered through `show()` for visual inspection. Because the kernel persists, later code can reuse earlier masks, reconstructions, plots, and partial analyses without recomputing them. The agent can therefore revise the analysis after seeing a wrong mask, an implausible trajectory, or a depth distribution that does not support the intended comparison. This feedback is appended to the model context, making intermediate results accessible at any subsequent step. The loop then returns to Stage II unless the agent has submitted an answer or the step count has reached the predefined maximum N_{max}.

Stage V: Answer submission. When the agent determines that it has collected sufficient evidence, it submits an answer with `ReturnAnswer()`. The loop terminates once the submitted answer conforms to the expected format for the question type (e.g., multiple choice, numerical, or free-form text). Otherwise, the loop continues to the next step.

4 Results

Evaluation setup. We evaluate on 20 spatial reasoning benchmarks spanning single-image spatial reasoning [40, 29, 21, 48], multi-view spatial reasoning [43, 50, 52], general spatial reasoning [13, 47, 23], video spatial & 4D reasoning [26, 46, 58, 1, 9, 55], and general video understanding [59, 24, 12, 41]. Evaluation details are provided in §B. We evaluate across several open-source VLM backbones, including the 122B-A10B and 397B-A17B variants of Qwen3.5 [31], the 35B-A3B and 27B variants of Qwen3.6 [32], the 27B and Flash-Next variants of Qwen3.8 [30], and the 31B and 26B-A4B variants of Gemma4 [14]. Due to the large number of benchmarks and backbones, we cap evaluation at $1,000$ samples for benchmarks exceeding that size; otherwise, all samples are used. $N_{max} = 30$ is used for all benchmarks. For all results in Tab. 1 and Tab. 2, the same hyperparameters, system prompts, and perception tools are used across all benchmarks and backbone models; further details are in the supplementary material.

Benchmark results. Tab. 1 reports results across 20 spatial reasoning benchmarks, spanning a broad range of tasks including video reasoning. SpatialClaw consistently improves over the no-tool baseline across all eight backbone models, with the most pronounced gains on video spatial & 4D

Table 1: **Main results across 20 spatial reasoning benchmarks.** For each backbone, we compare a no-tool baseline (thinking mode without tool access) with our SpatialClaw agent framework.

Model	Method	*Single-image spatial reasoning*				*Multi-view spatial reasoning*			*General spatial reasoning*		
		ERQA [2025]	*Omni3D [2025]*	*OmniSpatial [2026]*	*SPBench [2025]*	*MindCube [2025]*	*MMSI [2025]*	*SPAR-Bench [2025]*	*BLINK [2024]*	*SpatialTree [2025]*	*ViewSpatial [2025]*
Qwen3.5-397B-A17B [31]	No-tool	63.0	57.0	62.1	54.0	58.3	46.2	57.2	72.9	65.1	57.3
	SpatialClaw	62.0 (−1.0)	55.1 (−1.9)	62.8 (+0.7)	63.3 (+9.3)	66.2 (+7.9)	49.7 (+3.5)	66.1 (+8.9)	76.7 (+3.8)	60.0 (−5.1)	57.3 (+0.0)
Qwen3.5-122B-A10B [31]	No-tool	61.0	55.1	43.3	52.2	49.7	42.7	52.6	73.2	61.6	52.8
	SpatialClaw	57.5 (−3.5)	52.3 (−2.8)	61.9 (+18.6)	63.6 (+11.4)	61.0 (+11.3)	45.3 (+2.6)	57.6 (+5.0)	74.6 (+1.4)	57.8 (−3.8)	54.5 (+1.7)
Qwen3.6-35B-A3B [33]	No-tool	57.2	51.2	57.0	52.4	45.4	38.0	47.8	70.6	58.3	52.0
	SpatialClaw	58.2 (+1.0)	54.0 (+2.8)	59.9 (+2.9)	63.0 (+10.6)	61.9 (+16.5)	45.9 (+7.9)	58.4 (+10.6)	74.1 (+3.5)	60.3 (+2.0)	52.9 (+0.9)
Qwen3.6-27B [32]	No-tool	59.5	55.9	60.8	58.0	51.2	40.9	55.0	70.5	62.6	53.9
	SpatialClaw	61.5 (+2.0)	57.7 (+1.8)	62.8 (+2.0)	64.8 (+6.8)	70.1 (+18.9)	53.9 (+13.0)	68.1 (+13.1)	77.4 (+6.9)	64.6 (+2.0)	59.1 (+5.2)
Qwen3.8-27B [30]	No-tool	67.8	42.7	67.1	56.3	72.2	48.9	62.9	78.7	69.6	57.2
	SpatialClaw	71.8 (+4.0)	50.1 (+7.4)	66.6 (−0.5)	67.5 (+11.1)	83.5 (+11.3)	64.7 (+15.8)	72.2 (+9.3)	81.3 (+2.6)	74.0 (+4.4)	64.1 (+6.9)
Qwen3.8-Flash-Next [30]	No-tool	70.0	45.7	70.5	59.1	76.7	50.1	63.8	81.0	70.2	61.3
	SpatialClaw	73.0 (+3.0)	51.9 (+6.2)	69.1 (−1.4)	65.1 (+6.0)	81.8 (+5.1)	65.8 (+15.7)	71.6 (+7.8)	82.0 (+1.0)	72.4 (+2.2)	64.4 (+3.1)
Gemma4-31B [14]	No-tool	58.3	51.7	57.3	55.1	57.5	37.9	55.2	75.7	59.9	51.7
	SpatialClaw	61.3 (+3.0)	54.3 (+2.6)	63.6 (+6.3)	68.4 (+13.3)	72.8 (+15.3)	51.3 (+13.4)	63.3 (+8.1)	73.4 (−2.3)	60.7 (+0.8)	60.2 (+8.5)
Gemma4-26B-A4B [14]	No-tool	56.0	44.8	56.9	45.1	47.8	31.5	48.3	70.2	53.7	53.1
	SpatialClaw	56.0 (+0.0)	48.9 (+4.1)	57.1 (+0.2)	61.9 (+16.8)	64.0 (+16.2)	42.7 (+11.2)	63.1 (+14.8)	71.5 (+1.3)	57.9 (+4.2)	58.6 (+5.5)

Model	Method	*Video spatial & 4D reasoning*						*General video understanding*				*Average*
		MMSI-Video [2025]	*OSI-Bench [2025]*	*PAI-Bench [2025]*	*VSI-Bench-U [2025]*	*VSTI-Bench [2026]*	*DSI-Bench [2025]*	*CV-Bench [2025]*	*PerceptComp [2026]*	*Video-MME [2025]*	*Video-MME-v2 [2026]*	**Avg.**
Qwen3.5-397B-A17B [31]	No-tool	40.8	37.3	73.3	58.8	60.4	49.4	71.7	42.1	77.5	41.1	**57.3**
	SpatialClaw	43.6 (+2.8)	45.6 (+8.3)	71.6 (−1.7)	56.6 (−2.2)	67.5 (+7.1)	65.8 (+16.4)	72.9 (+1.2)	43.0 (+0.9)	77.0 (−0.5)	45.7 (+4.6)	**60.4** (+3.1)
Qwen3.5-122B-A10B [31]	No-tool	37.9	37.1	69.8	55.7	54.7	47.5	72.1	41.6	75.8	38.2	**53.7**
	SpatialClaw	38.3 (+0.4)	41.4 (+4.3)	65.6 (−4.2)	50.6 (−5.1)	63.3 (+8.6)	66.4 (+18.9)	72.2 (+0.1)	41.1 (−0.5)	72.6 (−3.2)	40.2 (+2.0)	**56.9** (+3.2)
Qwen3.6-35B-A3B [33]	No-tool	35.6	35.5	67.6	54.2	58.8	47.2	71.1	40.4	74.3	37.0	**52.6**
	SpatialClaw	38.5 (+2.9)	41.9 (+6.4)	63.8 (−3.8)	51.7 (−2.5)	64.2 (+5.4)	64.1 (+16.9)	74.0 (+2.9)	42.0 (+1.6)	75.0 (+0.7)	40.1 (+3.1)	**57.2** (+4.6)
Qwen3.6-27B [32]	No-tool	37.1	36.8	69.8	55.4	60.5	48.0	70.6	39.4	76.7	36.7	**55.0**
	SpatialClaw	47.0 (+9.9)	47.9 (+11.1)	72.1 (+2.3)	57.9 (+2.5)	70.3 (+9.8)	68.1 (+20.1)	75.1 (+4.5)	46.5 (+7.1)	79.4 (+2.7)	49.7 (+13.0)	**62.7** (+7.7)
Qwen3.8-27B [30]	No-tool	43.2	38.4	74.6	60.9	63.6	59.7	72.1	41.0	77.3	41.6	**59.8**
	SpatialClaw	52.6 (+9.4)	48.4 (+10.1)	74.9 (+0.3)	65.9 (+4.9)	79.1 (+15.6)	74.2 (+14.5)	75.4 (+3.3)	49.2 (+8.2)	79.3 (+2.0)	50.5 (+8.9)	**67.3** (+7.5)
Qwen3.8-Flash-Next [30]	No-tool	43.9	40.8	78.7	61.4	64.2	62.7	73.5	43.9	79.2	43.6	**62.0**
	SpatialClaw	55.1 (+11.2)	48.0 (+7.2)	76.7 (−2.0)	68.9 (+7.6)	81.2 (+17.0)	73.9 (+11.2)	78.5 (+5.0)	53.4 (+9.5)	82.5 (+3.3)	53.3 (+9.7)	**68.4** (+6.4)
Gemma4-31B [14]	No-tool	36.9	35.6	65.0	48.0	54.7	45.3	69.8	36.8	74.8	40.7	**53.4**
	SpatialClaw	41.6 (+4.7)	41.9 (+6.3)	68.1 (+3.1)	48.5 (+0.5)	67.6 (+12.9)	62.9 (+17.6)	72.2 (+2.4)	44.0 (+7.2)	77.0 (+2.2)	44.4 (+3.7)	**59.9** (+6.5)
Gemma4-26B-A4B [14]	No-tool	32.5	31.1	59.1	30.1	53.1	41.1	66.3	34.8	69.5	35.6	**48.0**
	SpatialClaw	32.8 (+0.3)	40.0 (+8.9)	59.0 (−0.1)	38.5 (+8.4)	61.7 (+8.6)	61.1 (+20.0)	67.8 (+1.5)	36.3 (+1.5)	69.6 (+0.1)	37.3 (+1.7)	**54.3** (+6.3)

reasoning (e.g., DSI-Bench, avg. $+17.0\%p$) and multi-view spatial reasoning (e.g., MindCube, avg. $+12.8\%p$), categories that benefit most from iterative multi-step geometric computation across frames or viewpoints. Notably, these gains hold consistently across models ranging from 26B to 397B parameters without any modification, suggesting that our design generalizes to future models *without model-specific tuning*.

Action interface comparison. Tab. 2 compares SpatialClaw against two alternative action interfaces that share the same toolset and system prompt, differing only in the response format and the system prompt section describing the action interface. We also include a *No-tool* baseline that receives visual inputs and reasons directly without tool access. Implementation details for each baseline are provided in §D. *Single-Pass Code* moves to open code, but asks the model to generate one complete program before seeing execution feedback; *Structured Tool-Call* adds external perception through a JSON command interface. The results show that SpatialClaw consistently outperforms the other two action interfaces across all benchmarks, with the largest gains on tasks that require multi-step geometric composition. These results validate the generalizability of SpatialClaw's action interface for spatial reasoning.

Table 2: **Action interface comparison.** All variants use the same toolset. "Structured Tool-Call" exposes the tools through a JSON command interface [6]. "Single-Pass Code" generates one complete program before execution.

Benchmark	No-tool Baseline	Single-Pass Code	Structured Tool-Call	SpatialClaw (Ours)
Single-image spatial reasoning				
ERQA [40]	58.3	58.3	59.0	**61.3**
Omni3D [29]	51.7	48.2	**55.7**	54.3
OmniSpatial [21]	57.3	60.0	59.6	**63.6**
SPBench [48]	55.1	58.2	60.5	**68.4**
Multi-view spatial reasoning				
MindCube [43]	57.5	57.5	62.4	**72.8**
MMSI [50]	37.9	42.3	43.0	**51.3**
SPAR-Bench [52]	55.2	61.1	58.7	**63.3**
Video spatial & 4D reasoning				
MMSI-Video [26]	36.9	37.1	35.1	**41.6**
OSI-Bench [46]	35.6	36.8	40.3	**41.9**
PAI-Bench [58]	65.0	64.2	65.6	**68.1**
VSI-Bench-U [1]	48.0	47.8	**50.1**	48.5
VSTI-Bench [9]	54.7	64.2	63.5	**67.6**
DSI-Bench [55]	45.3	57.9	58.4	**62.9**
General spatial reasoning				
BLINK [13]	**75.7**	73.9	73.9	73.4
SpatialTree [47]	59.9	58.9	57.7	**60.7**
ViewSpatial [23]	51.7	52.0	55.5	**60.2**
General video understanding				
CV-Bench [59]	69.8	71.3	**73.6**	72.2
PerceptComp [24]	36.8	39.2	42.5	**44.0**
Video-MME [12]	74.8	74.6	75.8	**77.0**
Video-MME-v2 [41]	40.7	40.2	44.0	**44.4**
Average (20 bench.)	53.4	55.2	56.7	**59.9**

Table 3: **Comparison with other spatial agents.** All use the same Gemma4-31B [14] backbone model. *: video or multi-image not supported.

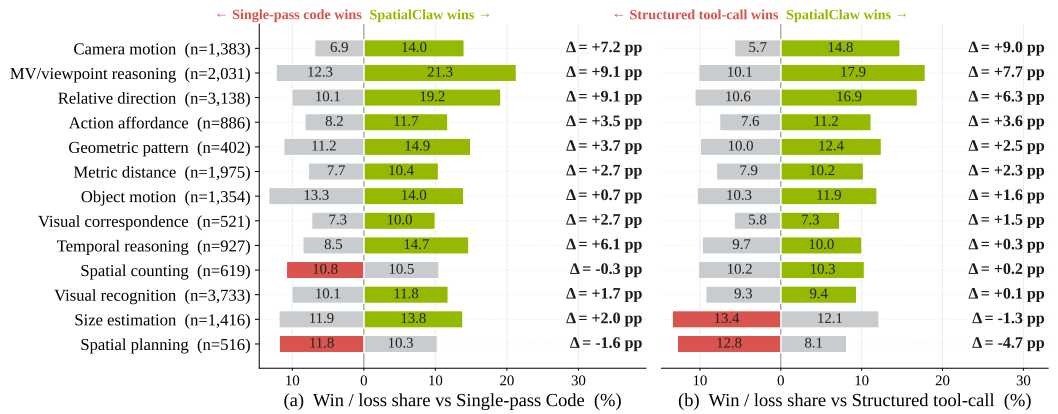
Benchmark	VADAR [29]	pySpatial [28]	SpaceTools Toolshed [6]	SpatialClaw (Ours)
Single-image spatial reasoning				
ERQA [40]	33.3	50.8	52.0	**61.3**
Omni3D [29]	44.6	50.6	50.7	**54.3**
OmniSpatial [21]	42.4	58.0	60.4	**63.6**
SPBench [48]	41.6	53.5	45.1	**68.4**
Multi-view spatial reasoning				
MindCube [43]	–*	67.1	52.9	**72.8**
MMSI [50]	–*	33.2	33.1	**51.3**
SPAR-Bench [52]	–*	51.7	53.9	**63.3**
Video spatial & 4D reasoning				
MMSI-Video [26]	–*	28.7	36.6	**41.6**
OSI-Bench [46]	–*	32.0	30.2	**41.9**
PAI-Bench [58]	–*	46.0	65.1	**68.1**
VSI-Bench-U [1]	–*	44.4	33.6	**48.5**
VSTI-Bench [9]	–*	55.0	56.0	**67.6**
DSI-Bench [55]	–*	43.7	43.0	**62.9**
General spatial reasoning				
BLINK [13]	–*	60.3	58.2	**73.4**
SpatialTree [47]	–*	53.9	52.7	**60.7**
ViewSpatial [23]	–*	49.9	52.1	**60.2**
General video understanding				
CV-Bench [59]	–*	67.7	46.8	**72.2**
PerceptComp [24]	–*	37.1	38.6	**44.0**
Video-MME [12]	–*	50.3	74.6	**77.0**
Video-MME-v2 [41]	–*	23.0	37.4	**44.4**
Average (20 bench.)	–*	47.8	48.7	**59.9**

Table 4: **Analysis on tools of SpatialClaw.** We ablate two design choices: No utility function, keeping only the perception tools (SAM3/DA3) and removing perception tools and keeping only the utility functions. We additionally report a no-tool baseline (backbone with thinking only). Each variant is evaluated on subsampled examples (500 per benchmark) across 15 benchmarks due to computation constraints. Gemma4-26B-A4B is used as the backbone for its small size.

Variant	*Single-image spatial reasoning*				*Multi-view spatial reasoning*			*General spatial reasoning*			*Video spatial & 4D reasoning*				*General video understanding*	*Average*
	ERQA [2025]	Omni3D [2025]	OmniSpatial [2026]	SPBench [2025]	MindCube [2025]	MMSI [2025]	SPAR-Bench [2025]	BLINK [2024]	SpatialTree [2025]	ViewSpatial [2025]	OSI-Bench [2025]	PAI-Bench [2025]	VSTI-Bench [2026]	DSI-Bench [2025]	CV-Bench [2025]	**Avg.**
SpatialClaw (Full)	56.0	36.6	57.2	**65.2**	67.2	**45.1**	**67.7**	**71.9**	**37.8**	**60.6**	**44.2**	**56.2**	58.3	61.0	**68.9**	**56.9**
(I) No utility functions (e.g., `tools.Mask`, `tools.Geometry`)	52.5	**37.0**	61.6	62.4	**69.0**	43.1	66.1	**71.9**	35.1	60.4	42.8	53.0	**59.9**	**62.0**	68.7	56.4
(II) No perception tools (no SAM3/DA3)	**58.0**	33.8	**62.2**	58.4	49.0	35.5	53.8	70.1	33.1	56.8	34.0	55.2	53.6	48.8	**68.9**	51.4
No-tool baseline	55.8	32.8	56.6	47.0	47.6	31.5	50.8	69.1	35.5	54.8	35.0	56.0	49.6	43.6	65.4	48.7

Quantitative comparison with other spatial agent methods. Tab. 3 compares SpatialClaw against recent spatial agent methods [29, 28, 6] using the same Gemma4-31B [14] backbone for all methods, using the official implementation of each method. Here, VADAR and pySpatial fall into the category of single-pass code, while SpaceTools uses the structured tool-call interface. VADAR does not support video or multi-image inputs, so the corresponding entries are left blank.

Overall, SpatialClaw outperforms all baselines across all benchmarks, achieving the largest margin over SpaceTools ($+11.2\%p$ on average). Among the baselines, SpaceTools ranks highest, consistent



(a) Win / loss share vs Single-pass Code (%) (b) Win / loss share vs Structured tool-call (%)

Figure 4: **Pairwise win/loss margin of SpatialClaw over baselines across 13 meta-categories.** SpatialClaw outperforms both (a) Structured tool-call and (b) Single-pass Code in 11/13 categories. The largest gains concentrate in categories that demand multi-step geometric composition.

with Tab. 2, which shows that structured tool-calls generally outperform single-pass code. SpatialClaw further improves over both action interfaces across all evaluated setups.

We also note that the baselines do not consistently improve over the no-tool baseline. For SpaceTools, we hypothesize that the method is designed to be fine-tuned via reinforcement learning, which may limit its zero-shot generalization. Other baselines may similarly have been optimized for narrower task categories, which could partly explain their limited performance across the diverse benchmarks evaluated here.

Generalization to proprietary backbones. To test whether SpatialClaw transfers beyond open-weight models, we further evaluate four proprietary backbones: Claude Opus 5, GPT-6 Astra, GPT-6 Sol, and GPT-6 Luna. Due to API costs, we sample 100 examples per benchmark using the same sampling procedure as in §H.1, and report the mean score across all 20 benchmarks. As shown in Tab. 5, the same harness improves every backbone without model-specific tuning, with gains ranging from $+2.1\%p$ (GPT-6 Luna) to $+15.0\%p$ (Claude Opus 5). Notably, even the strongest no-tool backbone, GPT-6 Astra, still benefits from SpatialClaw, reaching 77.4%.

Table 5: **SpatialClaw with proprietary backbones.** Mean score over 20 benchmarks.

Backbone	No-tool	SpatialClaw
Proprietary (100 samples / benchmark)		
Claude Opus 5	58.5	**73.5** (+15.0)
GPT-6 Astra	71.3	**77.4** (+6.1)
GPT-6 Sol	61.2	**72.9** (+11.7)
GPT-6 Luna	57.0	**59.0** (+2.1)

5 Analysis and Insights

In this section, we investigate *why* SpatialClaw's action interface improves spatial reasoning beyond what structured alternatives can achieve.

Finding 1. SpatialClaw generalizes across diverse spatial reasoning tasks even without pre-defined utility tools.

Tab. 4 reports an ablation study examining how the composition of the tool set affects performance. In variant **(I)**, we remove all utility wrappers (e.g., `tools.Mask`, `tools.Geometry`) described in §G, retaining only the core perception tools (SAM3/DA3) and the scientific computing libraries (`numpy`, `scipy`) available in the execution environment. This configuration tests whether the agent can substitute pre-defined utility logic with on-the-fly numerical computation. Variant **(I)** achieves performance on par with full SpatialClaw, suggesting that the persistent kernel with scientific primitives can largely compensate for the absent utility tools. We also report a no-perception variant **(II)**, which removes the perception tools (SAM3/DA3) from full SpatialClaw, leaving only the

code-as-action interface with scientific libraries. The resulting $+2.7\%p$ gain over the no-tool baseline isolates the contribution of the action interface itself, independent of the perception tools.

Finding 2. The agent spontaneously adapts its tool composition to the question type: distance questions preferentially invoke KD-tree search and norm operations, while direction questions rely on dot products. (Fig. 5)

To understand *how* the agent composes tools, we analyze the distribution of primitives (i.e., numpy and scipy operations) invoked across meta-categories (Figure 5). To enable a unified analysis across benchmarks, we group the fine-grained task categories of the 20 benchmarks into 13 semantically coherent meta-categories (e.g., depth estimation, relative direction, camera motion). The heatmap reveals a clear specialization: distance-type questions heavily use spatial indexing (`KDTree`) and vector norms, which are the natural building blocks for nearest-neighbor and proximity; direction-type questions instead rely on dot products and angular operations, which capture orientation and heading. Critically, this specialization is *not* hard-coded; no category-specific prompt engineering or tool routing was applied. The agent selects geometrically appropriate primitives solely from the question semantics, demonstrating that SpatialClaw's action interface unlocks a form of spontaneous, task-adaptive composition that is difficult to achieve with a structured tool-call interface.

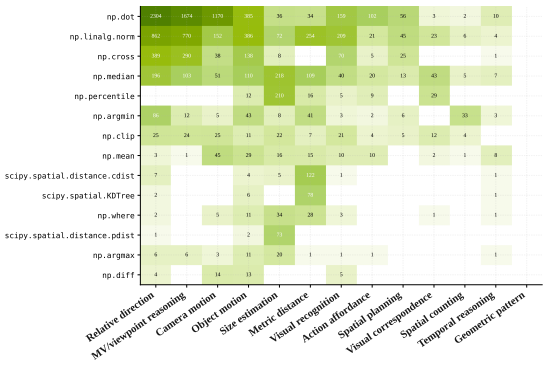


Figure 5: **Composition adapts to the question type.** Primitive usage frequency across meta-categories.

Finding 3. SpatialClaw gains are largest precisely where chained geometric computation across frames and viewpoints is required, validating SpatialClaw's action interface design. (Fig. 4)

In Figure 4, we compare SpatialClaw against the Structured tool-call and Single-pass Code baselines across 13 meta-categories. To construct these meta-categories, we map each benchmark's fine-grained category labels into one of 13 coarser groups spanning all 20 evaluated benchmarks. SpatialClaw secures a net advantage in 11/13 categories over both Structured tool-call and Single-pass Code. The largest lifts (+6–9 pp) concentrate in Camera motion, Multi-view/viewpoint reasoning, and Relative direction, precisely the categories that require chained geometric computation across frames and viewpoints. This is where the persistent kernel, by enabling cross-step composition and revision, provides the greatest leverage. Where gains are smaller, the bottleneck is perception quality: Visual recognition tasks are already near-saturated by the backbone VLM, leaving little room for interface-level gains. Together, this breakdown confirms that the expressive action interface is the primary driver of performance, not model capacity or tool coverage.

6 Conclusion

We presented SpatialClaw, a training-free spatial reasoning agent that adopts code as the action interface, enabling a VLM to flexibly compose, inspect, and revise perception tool outputs across steps in a persistent Python kernel. Evaluated across 20 spatial reasoning benchmarks and eight VLM backbones from two model families, SpatialClaw achieves 59.9% average accuracy and outperforms the recent spatial agent by **+11.2 points**, without any model- or benchmark-specific adaptation. These results demonstrate that the design of the action interface is a highly impactful yet underexplored axis of improvement for spatial reasoning agents.

Acknowledgments and Disclosure of Funding

We are deeply grateful to Valts Blukis, Johan Björck, Yulong Cao, Siyi Chen, Yi Dong, Jiaojiao Fan, Arushi Goel, Ehsan Hosseini Asl, Ehsan Jahangiri, Lily Lee, Zhiqi Li, Tsung-Yi Lin, Karan Sapra, Andrew Tao, Bowen Wen, Chiao-An Yang, Hanrong Ye, Zhiding Yu, and Haotian Zhang for their valuable discussions, insightful feedback, and generous support throughout this work. This work was funded by NVIDIA. The authors declare no competing interests.

References

[1] Ellis Brown, Jihan Yang, Shusheng Yang, Rob Fergus, and Saining Xie. Benchmark designers should "train on the test set" to expose exploitable non-visual shortcuts. *ArXiv Preprint*, 2025.

[2] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *European Conference on Computer Vision (ECCV)*, 2020.

[3] Nicolas Carion, Laura Gustafson, Yuan-Ting Hu, Shoubhik Debnath, Ronghang Hu, Didac Suris, Chaitanya Ryali, Kalyan Vasudev Alwala, Haitham Khedr, Andrew Huang, et al. Sam 3: Segment anything with concepts. In *International Conference on Learning Representations (ICLR)*, 2026.

[4] Boyuan Chen, Zhuo Xu, Sean Kirmani, Brain Ichter, Dorsa Sadigh, Leonidas Guibas, and Fei Xia. Spatialvlm: Endowing vision-language models with spatial reasoning capabilities. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

[5] Guo Chen, Zhiqi Li, Shihao Wang, Jindong Jiang, Yicheng Liu, Lidong Lu, De-An Huang, Wonmin Byeon, Matthieu Le, Max Ehrlich, Tong Lu, Limin Wang, Bryan Catanzaro, Jan Kautz, Andrew Tao, Zhiding Yu, and Guilin Liu. Eagle 2.5: Boosting long-context post-training for frontier vision-language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[6] Siyi Chen, Mikaela Angelina Uy, Chan Hee Song, Faisal Ladhak, Adithyavairavan Murali, Qing Qu, Stan Birchfield, Valts Blukis, and Jonathan Tremblay. Spacetools: Tool-augmented spatial reasoning via double interactive rl. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2026.

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

[8] Seokju Cho, Abhishek Badki, Hang Su, Jindong Jiang, Ziyao Zeng, Seungryong Kim, Sifei Liu, and Orazio Gallo. 4dp-qa: Scalable qa for 4d perception in vision language models. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2026.

[9] Zhiwen Fan, Jian Zhang, Renjie Li, Junge Zhang, Runjin Chen, Hezhen Hu, Kevin Wang, Huaizhi Qu, Dilin Wang, Zhicheng Yan, et al. Vlm-3r: Vision-language models augmented with instruction-aligned 3d reconstruction. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2026.

[10] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[11] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training compression for generative pretrained transformers. *ArXiv Preprint*, 2022.

[12] Chaoyou Fu, Yuhan Dai, Yongdong Luo, Lei Li, Shuhuai Ren, Renrui Zhang, Zihan Wang, Chenyu Zhou, Yunhang Shen, Mengdan Zhang, et al. Video-mme: The first-ever comprehensive evaluation benchmark of multi-modal llms in video analysis. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025.

[13] Xingyu Fu, Yushi Hu, Bangzheng Li, Yu Feng, Haoyu Wang, Xudong Lin, Dan Roth, Noah A Smith, Wei-Chiu Ma, and Ranjay Krishna. Blink: Multimodal large language models can see but not perceive. In *European Conference on Computer Vision (ECCV)*, 2024.

[14] Google DeepMind. Gemma 4. `https://deepmind.google/models/gemma/gemma-4/`, 2026. Accessed: 2026-04-14.

[15] Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

[16] Yi Han, Enshen Zhou, Shanyu Rong, Jingkun An, Pengwei Wang, Zhongyuan Wang, Cheng Chi, Lu Sheng, and Shanghang Zhang. Tiger: Tool-integrated geometric reasoning in vision-language models for robotics. *ArXiv Preprint*, 2025.

[17] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL `https://doi.org/10.1038/s41586-020-2649-2`.

[18] Wenbo Hu, Jingli Lin, Yilin Long, Yunlong Ran, Lihan Jiang, Yifan Wang, Chenming Zhu, Runsen Xu, Tai Wang, and Jiangmiao Pang. G^2vlm: Geometry grounded vision language model with unified 3d reconstruction and spatial reasoning. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2026.

[19] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9 (3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[20] Aastha Jhunjhunwala, Aayushi Ashokkumar Srivastava, Abhibha Gupta, Abhinav Garg, Abhinav Khattar, Achsah Wildish, Adam Rajfer, Adi Renduchintala, Adil Asif, Adriana Flores Miranda, et al. Nemotron 3.5 super and lightning: Open, efficient mixture-of-experts hybrid mamba-transformer models for agentic ai. *arXiv preprint arXiv:2610.38095*, 2026.

[21] 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. In *International Conference on Learning Representations (ICLR)*, 2026.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[23] Dingming Li, Hongxing Li, Zixuan Wang, Yuchen Yan, Hang Zhang, Siqi Chen, Guiyang Hou, Shengpei Jiang, Wenqi Zhang, Yongliang Shen, Weiming Lu, and Yueting Zhuang. Viewspatial-bench: Evaluating multi-perspective spatial localization in vision-language models. *ArXiv Preprint*, 2025.

[24] Shaoxuan Li, Zhixuan Zhao, Hanze Deng, Zirun Ma, Shulin Tian, Zuyan Liu, Yushi Hu, Haoning Wu, Yuhao Dong, Benlin Liu, Ziwei Liu, and Ranjay Krishna. PerceptionComp: A video benchmark for complex perception-centric reasoning. *ArXiv Preprint*, 2026.

[25] Haotong Lin, Sili Chen, Jun Hao Liew, Donny Y. Chen, Zhenyu Li, Guang Shi, Jiashi Feng, and Bingyi Kang. Depth anything 3: Recovering the visual space from any views. In *International Conference on Learning Representations (ICLR)*, 2026.

[26] Jingli Lin, Runsen Xu, Shaohao Zhu, Sihan Yang, Peizhou Cao, Yunlong Ran, Miao Hu, Chenming Zhu, Yiman Xie, Yilin Long, Wenbo Hu, Dahua Lin, Tai Wang, and Jiangmiao Pang. Mmsi-video-bench: A holistic benchmark for video-based spatial intelligence. *ArXiv Preprint*, 2025.

[27] Pan Lu, Bowen Chen, Sheng Liu, Rahul Thapa, Joseph Boen, and James Zou. Octotools: An agentic framework with extensible tools for complex reasoning. In *Proceedings of the Association for Computational Linguistics (ACL)*, 2026.

[28] 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. In *International Conference on Learning Representations (ICLR)*, 2026.

[29] Damiano Marsili, Rohun Agrawal, Yisong Yue, and Georgia Gkioxari. Visual agentic ai for spatial reasoning with a dynamic api. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025.

[30] Qwen Team. On the design of Qwen3.8-Next architecture: Evaluation, efficiency, and training stability. Technical report, Alibaba Group, August 2026.

[31] Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

[32] Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026. URL `https://qwen.ai/blog?id=qwen3.6-27b`.

[33] Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026. URL `https://qwen.ai/blog?id=qwen3.6-35b-a3b`.

[34] 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. In *International Conference on Learning Representations (ICLR)*, 2025.

[35] Fernando Ropero, Erkin Turkoz, Daniel Matos, Junqing Du, Antonio Ruiz, Yanfeng Zhang, Lu Liu, Mingwei Sun, and Yongliang Wang. Riemind: Geometry-grounded spatial agent for scene understanding. *ArXiv Preprint*, 2026.

[36] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[37] Chan Hee Song, Valts Blukis, Jonathan Tremblay, Stephen Tyree, Yu Su, and Stan Birchfield. RoboSpatial: Teaching spatial understanding to 2D and 3D vision-language models for robotics. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025.

[38] Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. In *IEEE International Conference on Computer Vision (ICCV)*, 2023.

[39] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *ArXiv Preprint*, 2023.

[40] Gemini Robotics Team, Saminda Abeyruwan, Joshua Ainslie, Jean-Baptiste Alayrac, Montserrat Gonzalez Arenas, Travis Armstrong, Ashwin Balakrishna, Robert Baruch, Maria Bauza, Michiel Blokzijl, et al. Gemini robotics: Bringing ai into the physical world. *ArXiv Preprint*, 2025.

[41] Video-MME Team. Video-mme-v2: Evaluating true understanding and reasoning in video mllms. *ArXiv Preprint*, 2026. URL `https://github.com/Video-MME/Video-MME-v2`.

[42] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

[43] Qineng Wang, Baiqiao Yin, Pingyue Zhang, Jianshu Zhang, Kangrui Wang, Zihan Wang, Jieyu Zhang, Keshigeyan Chandrasegaran, Han Liu, Ranjay Krishna, Saining Xie, Manling Li, Jiajun Wu, and Li Fei-Fei. Spatial mental modeling from limited views. *ArXiv Preprint*, 2025.

[44] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *International Conference on Machine Learning (ICML)*, 2024.

[45] Yifan Wang, Jianjun Zhou, Haoyi Zhu, Wenzheng Chang, Yang Zhou, Zizun Li, Junyi Chen, Jiangmiao Pang, Chunhua Shen, and Tong He. π^3: Permutation-equivariant visual geometry learning. In *International Conference on Learning Representations (ICLR)*, 2026.

[46] Mingrui Wu, Zhaozhi Wang, Fangjinhua Wang, Jiaolong Yang, Marc Pollefeys, and Tong Zhang. From indoor to open world: Revealing the spatial reasoning gap in mllms. *ArXiv Preprint*, 2025.

[47] Yuxi Xiao, Longfei Li, Shen Yan, Xinhang Liu, Sida Peng, Yunchao Wei, Xiaowei Zhou, and Bingyi Kang. Spatialtree: How spatial abilities branch out in mllms. *ArXiv Preprint*, 2025.

[48] Peiran Xu, Sudong Wang, Yao Zhu, Jianing Li, and Yunjian Zhang. Spatialbench: Benchmarking multimodal large language models for spatial cognition. *ArXiv Preprint*, 2025.

[49] Jihan Yang, Shusheng Yang, Anjali Gupta, Rilyn Han, Li Fei-Fei, and Saining Xie. Thinking in Space: How Multimodal Large Language Models See, Remember and Recall Spaces. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025.

[50] Sihan Yang, Runsen Xu, Yiman Xie, Sizhe Yang, Mo Li, Jingli Lin, Chenming Zhu, Xiaochen Chen, Haodong Duan, Xiangyu Yue, Dahua Lin, Tai Wang, and Jiangmiao Pang. Mmsi-bench: A benchmark for multi-image spatial intelligence. In *ICLR*, 2025.

[51] Chen Zeren, Lu Xiaoya, Zheng Zhijie, Li Pengrui, He Lehan, Zhou Yijin, Shao Jing, Zhuang Bohan, and Sheng Lu. Geometrically-constrained agent for spatial reasoning. *ArXiv Preprint*, 2025.

[52] Jiahui Zhang, Yurui Chen, Yanpeng Zhou, Yueming Xu, Ze Huang, Jilin Mei, Junhui Chen, Yujie Yuan, Xinyue Cai, Guowei Huang, Xingyue Quan, Hang Xu, and Li Zhang. From flatland to space: Teaching vision-language models to perceive and reason in 3d. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[53] Shihua Zhang, Qiuhong Shen, Shizun Wang, Tianbo Pan, and Xinchao Wang. Make geometry matter for spatial reasoning. *ArXiv Preprint*, 2026.

[54] Zaibin Zhang, Yuhan Wu, Lianjie Jia, Yifan Wang, Zhongbo Zhang, Yijiang Li, Binghao Ran, Fuxi Zhang, Zhuohan Sun, Zhenfei Yin, et al. Think3d: Thinking with space for spatial reasoning. *ArXiv Preprint*, 2026.

[55] Ziang Zhang, Zehan Wang, Guanghao Zhang, Weilong Dai, Yan Xia, Ziang Yan, Minjie Hong, and Zhou Zhao. Dsi-bench: A benchmark for dynamic spatial intelligence. *ArXiv Preprint*, 2025.

[56] Shitian Zhao, Haoquan Zhang, Shaoheng Lin, Ming Li, Qilong Wu, Kaipeng Zhang, and Chen Wei. Pyvision: Agentic vision with dynamic tooling. *ArXiv Preprint*, 2025.

[57] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[58] Fengzhe Zhou, Jiannan Huang, Jialuo Li, Deva Ramanan, and Humphrey Shi. Pai-bench: A comprehensive benchmark for physical ai. *ArXiv Preprint*, 2025.

[59] Nannan Zhu, Yonghao Dong, Teng Wang, Xueqian Li, Shengjun Deng, Yijia Wang, Zheng Hong, Tiantian Geng, Guo Niu, Hanyan Huang, et al. Cvbench: Benchmarking cross-video synergies for complex multimodal reasoning. *ArXiv Preprint*, 2025.

Supplementary Material

Contents

A Related Works **16**
B Evaluation Protocol **16**
C Additional Analysis **18**
D Baseline Implementations **19**
D.1 Single-Pass Code **19**
D.2 Structured Tool-Calls **19**
E Agent System Design **19**
E.1 System Configuration **19**
E.2 Input Preprocessing **20**
E.3 Persistent Kernel **20**
E.4 Security Sandbox **20**
E.5 Per-Frame Type Contract **20**
E.6 Error Handling **20**
E.7 Backbones **21**
F Prompt Details **22**
F.1 Main Agent Prompt **22**
F.2 Planner Prompt **23**
F.3 Vision Prompts **23**
G Tool API Reference **24**
G.1 `tools.Reconstruct` **24**
G.2 `tools.SAM3` **25**
G.3 `tools.Geometry` **26**
G.4 `tools.Mask` **26**
G.5 `tools.Time` **27**
G.6 `tools.Graph` **27**
G.7 `tools.Draw` **28**
H Additional Experiments and Analysis **29**
H.1 General-Purpose Coding-Agent Comparison **29**
H.2 Native Coding-Agent Comparison with a Proprietary Backbone **29**
H.3 Computation and Wall-Clock Latency **30**
I Limitations and Broader Impact **32**
I.1 Limitations **32**
I.2 Broader Impact **32**

A Related Works

Spatial reasoning in VLMs. Despite broad progress in vision-language models (VLMs), spatial reasoning remains a persistently limited capability [4, 7, 37]. A common response is to fine-tune VLMs with spatial supervision, either by distilling 3D annotations into instruction data [4, 7] or by augmenting the model with explicit geometry modules [18, 9, 53]. These approaches produce fast, self-contained models, but require retraining whenever perception modules or task distributions change.

Tool-augmented visual agents. Tool-augmented visual agents have an LLM compose calls to specialist vision modules [15, 38, 36, 56, 27]. Early work synthesizes a full program in a single pass [15, 38], subsequent work dispatches requests through structured tool menus [36, 27]. These systems establish that external perception can extend an LLM beyond its native visual capabilities, but they either constrain the agent to a fixed tool interface that may limit generalization to novel compositions, or commit to a full program before any intermediate output can be inspected, or fuse perception and planning inside a single multimodal model that cannot independently verify intermediate tool outputs.

Spatial reasoning agents. GCA [51] decouples the VLM into a semantic analyst that formalizes the query as geometric constraints and a task solver that executes tool calls within those bounds, RieMind [35] grounds an LLM in an explicit 3D scene graph queried through typed geometric operations, and SpaceTools [6] fine-tunes a VLM to coordinate a predefined set of perception tools through supervised demonstrations followed by interactive reinforcement learning. Think3D [54] uses a closely related but more specialized interface, repeatedly selecting camera viewpoints to render a reconstructed point cloud and reasoning over the rendered images. pySpatial [28] composes reconstruction, camera-pose recovery, and novel-view synthesis into a 3D visual program, and VADAR [29] first synthesizes a task-specific Pythonic API and then a program that calls into it. Neither writes code turn-by-turn in response to intermediate execution results, so per-step inspection of perception outputs is limited.

Code action interface for LLM agents. CodeAct [44] showed that emitting executable Python code as the action outperforms JSON- and text-formatted action spaces for general-purpose LLM agents. SpatialClaw instantiates this paradigm for spatial reasoning, contributing domain-specific design choices absent from general-purpose frameworks. Where CodeAct focuses on the infrastructure of code execution, our contribution is the spatial-reasoning angle, namely controlled comparisons against alternative action interfaces and trace-level analyses that identify when code action interface helps and which spatial analysis patterns drive the gains (§4, §5).

B Evaluation Protocol

We evaluate SpatialClaw on the 20 spatial reasoning benchmarks reported in the main results, organized into five categories: single-image spatial reasoning (ERQA [40], Omni3D [29], OmniSpatial [21], SPBench [48]), multi-view spatial reasoning (MindCube [43], MMSI [50], SPAR-Bench [52]), video spatial & 4D reasoning (MMSI-Video [26], OSI-Bench [46], PAI-Bench [58], VSI-Bench-U [1], VSTI-Bench [9], DSI-Bench [55]), general spatial reasoning (BLINK [13], SpatialTree [47], ViewSpatial [23]), and general video understanding (CV-Bench [59], PerceptComp [24], Video-MME [12], Video-MME-v2 [41]). Table 6 reports the per-sample scoring metric applied to each benchmark.

We apply a consistent per-sample scoring protocol across all benchmarks, using the same metric family while respecting each benchmark's original threshold settings. Categorical questions, including multiple-choice and binary questions, receive a score of 1 when the predicted answer matches the reference and 0 otherwise. Numerical questions are scored with mean relative accuracy (MRA) [49], where the acceptance threshold follows each benchmark's originating implementation. SPAR-Bench's view-change-inference task uses VCI, defined as MRA averaged over its five movement axes. The reported number for each benchmark is the unweighted mean of the per-sample scores.

Table 6: **Benchmark-specific evaluation metrics and scoring protocols.** Each benchmark mixes categorical questions (multiple-choice or binary) and numerical questions in different proportions. `Acc` denotes per-sample 1/0 categorical scoring, `MRA` denotes mean relative accuracy on numerical questions, and `VCI` denotes the SPAR-Bench metric used for the view change inference task (MRA averaged across five movement axes). The reported number for each benchmark is the unweighted mean of the per-sample scores.

Benchmark	**Category**	**Per-sample metric**
ERQA [40]	Single-image spatial reasoning	Acc
Omni3D [29]	Single-image spatial reasoning	Acc + MRA
OmniSpatial [21]	Single-image spatial reasoning	Acc
SPBench [48]	Single-image spatial reasoning	Acc + MRA
MindCube [43]	Multi-view spatial reasoning	Acc
MMSI [50]	Multi-view spatial reasoning	Acc
SPAR-Bench [52]	Multi-view spatial reasoning	Acc + MRA + VCI
MMSI-Video [26]	Video spatial & 4D reasoning	Acc
OSI-Bench [46]	Video spatial & 4D reasoning	Acc + MRA
PAI-Bench [58]	Video spatial & 4D reasoning	Acc
VSI-Bench-U [1]	Video spatial & 4D reasoning	Acc + MRA
VSTI-Bench [9]	Video spatial & 4D reasoning	Acc + MRA
DSI-Bench [55]	Video spatial & 4D reasoning	Acc
BLINK [13]	General spatial reasoning	Acc
SpatialTree [47]	General spatial reasoning	Acc + MRA
ViewSpatial [23]	General spatial reasoning	Acc
CV-Bench [59]	General video understanding	Acc
PerceptComp [24]	General video understanding	Acc
Video-MME [12]	General video understanding	Acc
Video-MME-v2 [41]	General video understanding	Acc

For benchmarks with more than $1,000$ samples, we evaluate a randomly chosen subset of $1,000$ samples drawn with a fixed seed; the same subset is reproduced by running the supplementary code with the same seed.

C Additional Analysis

Finding 4. Composition is the main driver of SpatialClaw's gains over structured tool-call. (Fig. 6)

To identify the main driver of SpatialClaw's gains over structured tool-call baseline, we examine the instances where SpatialClaw answers correctly but structured tool-call fails (Figure 6). For each such instance, we provide an LLM judge (Gemini-3.1-Pro [39]) with the full reasoning traces of both systems, the ground-truth answer, and the input images, and ask it to assign a binary label to each predefined attribution category; instances in which all categories receive a negative label are classified as *interface-neutral*.

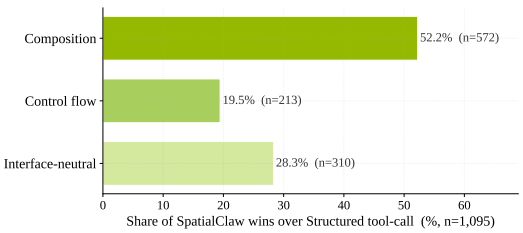


Figure 6: **Attribution of SpatialClaw's wins over structured tool-call via LLM-as-judge.** Over half of the gains are driven by *code composition*, 19.5% by *control flow*, and 28.3% are *interface-neutral* wins on perceptual tasks unaffected by the action interface.

We find that over 50% of the instances are attributed to *code composition* (i.e., chaining multiple tool calls into a single coherent program), while 19.5% are attributed to *control flow* (e.g., `if` or `for` statements that conditionally branch or iterate over intermediate results). The remaining 28.3% are *interface-neutral* wins, in which the correct answer depends on visual recognition or luck rather than the agent's action space, and either interface would have been equally capable. This is further causally supported by Tab. 4, which shows that performance degradation is minimal even when predefined utility functions are removed and replaced with on-the-fly numerical computation.

Finding 5. Geometric reasoning errors constitute a leading failure mode, with perception errors driven by VLM hallucinations and tool limitations as a notable secondary contributor.

In Fig. 7, we analyze the agent's failure modes by categorizing $1,000$ incorrect samples with LLM-as-Judge [57], using a strong proprietary model (Gemini-3.1-Pro). For each sample, the judge is provided with the agent's full reasoning trace and the ground-truth answer, and assigns each session to one of 11 human-designed fine-grained categories.

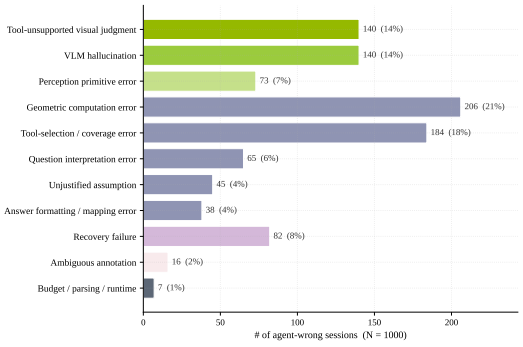


Figure 7: **Failure-mode breakdown of incorrect agent sessions.** Each session is classified by an LLM-as-Judge (Gemini-3.1-Pro [39]) into one of 11 fine-grained failure categories.

On the perception side, the agent frequently attempts visual judgments that exceed the capabilities of its available tools, and its underlying VLM occasionally hallucinates objects, attributes, or spatial relations that are not actually present in the scene. Lower-level perception primitives such as detection and segmentation also produce localized errors that propagate into downstream reasoning. The predominant reasoning failures are geometric in nature, as the agent frequently errs in handling 3D coordinates, distances, angles, or projective relationships, even when the underlying perceptual evidence is sound. Closely related is a class of tool-selection and coverage failures, in which the agent invokes an inappropriate tool or omits a necessary intermediate step. We additionally find cases of question misinterpretation, unjustified assumptions about unobserved scene properties, and inconsistencies between the internal computation and the final answer format. Beyond these direct errors, a non-trivial fraction of failures arises from the agent's inability to recover upon detecting an inconsistency, as it either commits to a flawed early hypothesis or oscillates without converging. The remaining residual

cases involve ambiguous ground-truth annotations and occasional budget, parsing, or runtime issues, which together account for only a small share of observed failures.

A promising direction for future work is to apply reinforcement learning to improve tool selection, coding of geometric operations, and error recovery within the agent loop.

D Baseline Implementations

§2 of the main paper compares the code as the action interface of SpatialClaw against two baselines, single-pass code and structured tool-calls. To isolate the effect of the interface, both baselines share the agent loop, the persistent kernel, the perception tools, and the planner; only the format of the per-step action differs, along with any adjustment to the per-sample step budget necessitated by the format.

D.1 Single-Pass Code

The single-pass baseline collapses the multi-step loop to a single iteration. The agent receives the question, the key frames, and the same main-agent system prompt, and must produce one Python cell that calls perception tools, performs any required computation, and submits the answer through `ReturnAnswer`. Planning is disabled, and the workflow section of the system prompt is replaced by a single-turn variant that informs the agent there will be no further opportunity to act. The agent must therefore commit to a complete strategy before observing any intermediate mask, depth map, plot, or runtime error. When the cell does not reach `ReturnAnswer`, the same termination fallback used by the code as the action interface variant (§E.6) returns a best-effort answer.

D.2 Structured Tool-Calls

The structured-tool-call baseline keeps the multi-step loop intact but constrains each step to a single named tool invocation expressed as JSON. The *Code* section of the response envelope is replaced by a *Tool Call* section that contains exactly one object of the form

```
{"tool": "tools.SAM3.segment_image_by_text",
 "args": {"image": "InputImages[0]", "prompt": "red car"}}
```

The specified tool call is first translated to Python code and then executed on the same kernel used by SpatialClaw, and its return value is stored in a named variable that later steps can reference. Arguments must be either literal values or references to previously bound names (i.e., the input variables, the tool catalogue, or the outputs of prior steps). Arbitrary Python expressions are not allowed, and each step issues exactly one tool call, and intermediate results are accessible only through these named per-step variables.

E Agent System Design

E.1 System Configuration

SpatialClaw runs on two independently scalable serving roles. The first role hosts the language-model backbone and serves every LLM call the system makes, including the main agent, the planner, and the two isolated visual sessions invoked from inside the kernel (`vlm.locate` for coordinate grounding and `vlm.ask_with_thinking` for visual reasoning). The backbone is served via vLLM [22]; multiple endpoints are reachable through a single OpenAI-compatible interface, and a session-aware router dispatches each call so that the prefix of the conversation lands on a consistent endpoint, exploiting the prefix cache that vLLM maintains across requests.

The second role hosts the perception models behind a lightweight HTTP service: Depth Anything 3 [25] for 3D reconstruction and SAM3 [3] for segmentation. The kernel-side tools `tools.Reconstruct` and `tools.SAM3` are light-weight clients that call this service and convert the results into ordinary numpy arrays before returning to the agent. For Depth Anything 3, we use the `DA3Nested-Giant-Large` variant, which pairs an any-view giant model with a metric depth model to reconstruct visual geometry at real-world metric scale.

This decomposition has two practical consequences. First, the language-model and perception backbones scale independently: the number of LLM nodes and perception nodes can each be tuned to their respective call demand without any changes to the agent code. Second, all LLM roles (agent, planner, grounding, and thinking) share the same backbone pool, so the main conversation and in-kernel visual queries share the same prefix cache and benefit from KV reuse.

E.2 Input Preprocessing

All input images are resized such that the long edge does not exceed 768 pixels, preserving the original aspect ratio; this bounds the per-frame token cost. The agent VLM receives at most 32 frames as visual context: for video inputs longer than 32 frames, frames are drawn at uniform temporal intervals to yield 32 key frames; for multi-image inputs, the sequence is truncated to the first 32 images. The persistent kernel retains the full sampled frame sequence, allowing any frame to be revisited via `show()` or forwarded to a perception tool in subsequent steps. When a sample contains multiple videos, each is maintained as an independent frame sequence, enabling the agent to analyze and compare them independently.

E.3 Persistent Kernel

Each example is solved within a dedicated, stateful IPython kernel that persists throughout the entire inference run for that sample. Variables created during one cell (segmentation masks, reconstructions, depth and point arrays, intermediate visualizations, and partial numerical results) remain in scope for every subsequent cell, so the agent can revisit and recompose earlier evidence without re-issuing the underlying tool call. A per-cell wall-clock timeout protects the system from runaway code, and on repeated execution failures the kernel is restarted; when this happens, the input frames, sample metadata, the `tools` module, the `feedback` module, and `ReturnAnswer` are automatically re-injected so the agent resumes against an identical environment.

E.4 Security Sandbox

Code emitted by the LLM is statically analyzed before execution. The analyzer traverses the Abstract Syntax Tree (AST) of each cell to identify and reject unsafe patterns, including file I/O, network access, dynamic-code primitives such as `exec` and `eval`, and direct imports of GPU-backend libraries. A complementary regular-expression pass catches patterns that may evade AST traversal, such as method-style writes (`.save`, `.to_csv`). When a cell is rejected, the corresponding error message is delivered to the agent as feedback so it can revise the code within the same step, without adding latency to steps that pass inspection.

E.5 Per-Frame Type Contract

Every frame-indexed perception output is encapsulated in a typed container that records the absolute frame indices to which it corresponds. Composing two such containers (e.g., a per-frame segmentation mask with a per-frame point map) triggers an automatic frame-index alignment check at the composition site; index mismatches raise an immediate exception identifying the offending frames. This mechanism guards against a class of silent semantic errors in which tensors are numerically well-formed but spatially misaligned, such as applying a mask derived from one frame to the point map of another. By surfacing the error at the point of composition, the violation appears directly in the agent's subsequent feedback rather than propagating silently to produce an incorrect final answer.

E.6 Error Handling

Adopting code as the action interface exposes the system to a broader class of runtime failures than a fixed-API interface. SpatialClaw treats each such failure as an additional observation by routing the resulting exception or traceback directly into the agent's next feedback message, enabling the agent to diagnose and revise its code within the same episode rather than terminating on error.

Response-format errors. If the agent's response cannot be parsed into the four required fields (*Purpose*, *Reasoning*, *Next Goal*, *Code*), the validator records a *format error* for the step. The malformed text is replaced in the conversation with a short placeholder that names the violation, and

the agent is reminded to re-read the required format on the next turn. The malformed body is never echoed back, both to keep the context clean and to discourage the agent from imitating it.

Sandbox rejections. Code that the security analyzer rejects (§E.4) is never executed. Instead, the rejection reason (identifying the forbidden module, builtin, or pattern) is returned as the step's error and surfaced to the agent in the next feedback message, exactly as if the code had failed at runtime. The agent therefore retries the same step with revised code rather than escalating.

Cell-execution exceptions. When a cell raises a Python exception, three steps are applied to trim the error output before the failure is forwarded to the agent. The following example illustrates the condensed representation that is inserted into the conversation history after a faulting step:

```
**Purpose**: Perform 3D reconstruction and extract point clouds for the fireplace and the painting.
**Reasoning**: [errored -- condensed]
**Next Goal**: [errored -- condensed]
**Code**:
```python
# 1. 3D Reconstruction
recon = tools.Reconstruct.Reconstruct(InputImages)

# 2. Extract point clouds
# Fireplace is in InputImages[1]
fi_fire = seg_fireplace.frame_indices[0] # <-- ERROR
# NameError: name 'seg_fireplace' is not defined
```
```

First, the traceback is truncated to the final exception type, its message, and the offending source line; surrounding interpreter frames that reflect the execution infrastructure rather than the agent's code are discarded. Second, any code appearing after the faulting line is excluded from the context window, since it was never executed and therefore contributes no evidence to the agent's subsequent reasoning. Third, the *Reasoning* and *Next Goal* fields of the failed step are replaced by a condensed summary prior to insertion into the conversation history, preventing a single failure from disproportionately consuming the available context budget.

Cell timeouts. Each cell is subject to a per-cell wall-clock timeout to guard against non-terminating or excessively long-running code. Upon timeout, the kernel's user namespace is cleared and the execution environment is restored by re-injecting the input frames, sample metadata, and the `tools`, `feedback`, and `ReturnAnswer` entry points; the agent receives a timeout notification in its next observation and is expected to revise its code accordingly. If namespace restoration fails, the kernel process is fully restarted and the same re-injection sequence is applied.

Perception-tool retries. Calls to the GPU-backed perception tools may fail transiently due to server restarts or temporary overload. Each `tools.Reconstruct` or `tools.SAM3` call is automatically retried up to a fixed number of times, with increasing wait intervals between attempts; each retry also re-selects an available endpoint to route away from an unhealthy server. Only if all retries are exhausted is the error propagated to the agent as a Python exception. Similarly, transient network errors on the language-model side (e.g., connection drops or gateway timeouts) are distinguished from agent-side failures and do not count against the agent's error budget.

Hard budgets and forced termination. The loop carries explicit budgets on the number of LLM steps, the number of consecutive step-level failures, and the cumulative number of tool calls. When any budget is exceeded, the loop hands control to a termination node that is guaranteed to return an answer. The termination node attempts two fallback strategies in order: (i) a chain-of-thought fallback that prompts the VLM backbone to answer directly from the key frames using the same boxed-answer protocol as the no-tool baseline; and (ii) if the chain-of-thought fallback does not yield a parseable answer, a regex-based extraction over the agent's recent messages and variables. The agent therefore always returns a best-effort answer for any sample whose execution started, even when the persistent-kernel path was unable to reach `ReturnAnswer` on its own.

E.7 Backbones

We evaluate SpatialClaw with the six open-source backbones listed in Table 7. All six are served through the same vLLM-based serving role described in §E.1. Crucially, every backbone is evaluated

Table 7: **Backbones used in the main results.** All six are served through the same vLLM-based [22] serving role and evaluated under identical configuration: the same system prompt, tool set, maximum step count, and input preprocessing across every benchmark.

Backbone	Size	Quantization	Context
Qwen3.5-397B-A17B [31]	397B (17B activated)	GPTQ Int4 [11]	262K
Qwen3.5-122B-A10B [31]	122B (10B activated)	FP8	262K
Qwen3.6-35B-A3B [33]	35B (3B activated)	FP8	262K
Qwen3.6-27B [32]	27B	FP8	262K
Gemma4-31B [14]	31B	FP8	256K
Gemma4-26B-A4B [14]	26B (4B activated)	FP8	256K

under *identical* configuration: the same system prompt, the same set of perception tools, the same maximum step count, and the same input preprocessing. No prompt template, tool subset, or budget is tuned per benchmark or per backbone.

F Prompt Details

SpatialClaw defines four VLM roles, each with its own system prompt: the main agent that writes code, the planner that produces a plan before any image is observed, and two isolated visual sessions invoked from inside the kernel through `vlm.locate` (coordinate grounding) and `vlm.ask_with_thinking` (visual reasoning). The number of calls per role varies per sample: the main agent is queried once per step, the planner is invoked once per sample, and the two kernel-invoked sessions are called as many times as the agent decides. This appendix summarizes what each prompt encodes.

F.1 Main Agent Prompt

The main agent's system prompt is organized into a fixed sequence of sections, which we describe in order. A short header introduces the role of the agent and names the input variables (`InputImages` and `Metadata`).

Response format. The first content section declares the response format: every reply must contain four markdown sections with the headings *Purpose*, *Reasoning*, *Next Goal*, and *Code*, where the code field holds a single Python cell to be executed in the kernel.

Visual access. The two following sections document the visual access surface available inside the kernel. The `show()` entry displays one or more images inline in the next feedback message, notes that `matplotlib` figures are auto-captured, and states the per-session image budgets. The `vlm.locate` entry calls an isolated grounding session that returns coordinates in a 0–1000 normalized scale and accepts up to eight images per call, and the `vlm.ask_with_thinking` entry calls an isolated reasoning session that returns a textual answer over up to 64 frames. The same section instructs the agent to treat `Not visible` and `Cannot determine from the images.` as valid responses and not to reassign the `vlm` or `feedback` variables.

Available tools. The *Available Tools* section then documents the seven tool namespaces that the kernel exposes, with method signatures, argument types, expected input shapes, and short usage examples for each: `tools.Reconstruct`, `tools.SAM3`, `tools.Graph`, `tools.Time`, `tools.Mask`, `tools.Geometry`, and `tools.Draw`. The details can be found in §G.

Coordinate systems. The coordinate-system section names four reference frames (pixel space, camera space, world space, and an object-centric frame defined by an object's facing direction), states the technical convention for camera-to-world matrices returned by `tools.Reconstruct` (4×4 SE(3) matrices in OpenCV convention, with gravity-aligned world axes anchored to the first camera), and gives a code snippet that computes left/right/front/behind classifications by projecting the world-frame vector to a target onto the camera's forward and right axes.

Robust computation. The robust-computation section lists five principles: prefer median over mean for aggregations, compare across multiple frames before drawing conclusions, reason in metric units rather than pixels, sanity-check magnitudes against physical priors, and print numerical values before committing to a conclusion.

Cross-validation. The cross-validation section enumerates four complementary evidence sources (visual perception via `show()`, geometric computation, visualizations, and logical reasoning) and specifies a five-step diagnostic procedure that the agent follows when two of them disagree.

Kernel-side contract and budget. The remaining sections fix the kernel-side contract and the run-level budget. The `ReturnAnswer` section documents the single-call API used to submit a final answer and the accepted argument types (`str`, `int`, `float`). The code-rules section lists the modules pre-imported into the kernel, the modules and built-ins forbidden by the security sandbox, and the reserved names that the agent must not reassign (`feedback`, `tools`, `InputImages`, `Metadata`, `ReturnAnswer`, `show`, `RefImages`). The workflow section instructs the agent to follow the planner's plan and to confirm any spatial conclusion against at least two independent lines of evidence before calling `ReturnAnswer`, and is followed by a single line stating the per-sample step budget. A final session-input section describes the per-sample input variables, the frame-indexing convention for `InputImages`, the fields of `Metadata`, and the key-frame-to-variable mapping; an additional reference-images section is included when the sample contains inline `[reference image #N]` tags.

F.2 Planner Prompt

The planner is invoked once per sample, before the main agent begins execution. The planner receives the question text, the per-sample metadata (frame count, frame indices, and frame-rate fields), and the system prompt only.

Shared documentation. After a one-paragraph header that declares the planner's role and states that it does not see the actual frames, the prompt repeats the same *Available Tools*, `show()`, `vlm.locate`, `vlm.ask_with_thinking`, and coordinate-system documentation that the main agent receives, and embeds the cross-validation and robust-computation sub-sections from the main agent prompt.

Planning strategy. The planning-strategy section maps question shapes to tool choices: questions about object coordinates point to `vlm.locate` followed by `tools.SAM3`; questions about three-dimensional geometry, depth, or camera pose point to `tools.Reconstruct`; questions answerable by visual judgment point to `show()`; and questions that require a textual reading over several frames, or a fallback when a specialized tool fails, point to `vlm.ask_with_thinking`. The same section gives explicit guidance on annotation overlays, on coordinate grounding through 0–1000 normalized pixel coordinates, and on the choice between quantitative computation and qualitative visual reading.

Task structure and rules. The session-input section reports the frame count and the frame-indexing convention and states that the planner cannot see the images, instructing it to plan an investigation rather than answer the question from the question text. The task section asks for six items in the output: a task analysis (including an explicit choice of coordinate system when the question is ambiguous), a list of information needs, an ordered computation plan, a verification checklist, a list of cross-validation steps, and a fallback plan. The critical-rules section states that the planner must respond in plain text only, must not include JSON outside the verification checklist, must not write executable Python code, and must not produce the answer or pre-conclude phrases such as "the answer is likely ..." or "I expect the answer is ..."; the planning-strategy section adds that any hypothesis the planner forms from the question is itself what the tools must verify, rather than evidence in its own right.

F.3 Vision Prompts

The two isolated visual sessions invoked from the kernel use their own system prompts.

Grounding session (`vlm.locate`). The grounding session called by `vlm.locate` specifies a three-step procedure: identify the exact object, annotation, or marker that the question describes;

classify the image as PRESENT, ABSENT, or AMBIGUOUS with respect to that description; and answer accordingly. PRESENT cases produce coordinates in the format requested by the calling question. ABSENT and AMBIGUOUS cases reply with the literal string `Not visible` on its own line, optionally followed by one short clarifying sentence. The prompt includes three worked examples that illustrate each of the three classifications, and ends with a short formatting block stating that coordinates use the 0–1000 normalized scale, that the assistant follows a "Reply with ONLY the numbers" instruction when present, and that the same prompt also covers segmentation-overlay assessment and plot-and-chart reading.

Reasoning session (`vlm.ask_with_thinking`). The reasoning session called by `vlm.ask_with_thinking` specifies five rules: 1) ground every claim in what is directly observable in the supplied images, 2) use any frame indices referenced by the question in the answer, 3) reason carefully across the frames before producing a concise final answer, 4) reply with the literal string `Cannot determine from the images.` together with a one-line note when the supplied frames do not constrain the answer, and 5) treat each call as independent of any prior context.

G Tool API Reference

This appendix expands the one-paragraph tool description in the main paper into per-tool signatures and behaviors. The first two tools (`tools.Reconstruct` and `tools.SAM3`) are thin clients that call the perception serving role described in §E.1; the remaining tools are pure-Python helper functions that execute in the same process as the kernel.

G.1 `tools.Reconstruct`

`tools.Reconstruct` is a client wrapper for the perception serving role described in §E.1. Its single entry method calls Depth Anything 3 [25] on a frame batch and returns a `Reconstruction` object that exposes per-frame depth, camera intrinsics, camera-to-world extrinsics, a dense per-frame point map in world coordinates, and a top-down rendering helper function. The output coordinate system is gravity-aligned with respect to the first camera: $+Y$ points up, the ground plane sits at $Y \approx 0$, and the first camera looks toward $-Z$. The output spatial resolution matches the input frame resolution, so masks from `tools.SAM3` can be combined directly with the reconstructed point maps without resizing.

Method.

- `Reconstruct(frames,frame_indices=None)`
  - *Input:* `frames` is a list of `FrameImage` or PIL (Python Imaging Library) images, with at most `reconstruct_max_frames` entries (set to 64 in our experiments). When `FrameImage` objects are passed, absolute video frame indices are auto-extracted from each frame's `frame_index` attribute and the `frame_indices` argument may be omitted.
  - *Output:* a `Reconstruction` object (described below).

Reconstruction attributes. The returned object is indexed by absolute frame index, not by position in the input list.

- `frame_indices`: `list[int]`, absolute frame indices in the same order as `frames`.
- `num_frames`: `int`; `metric_scale`: `float`.
- `depth[fi]`: `(H, W) float32` depth map at absolute frame `fi`.
- `extrinsics[fi]`: `(4, 4) float64` camera-to-world SE(3) matrix in OpenCV convention.
- `intrinsics[fi]`: dictionary with keys `fx`, `fy`, `cx`, `cy`.
- `points[fi]`: `(H, W, 3) float32` dense point map in world coordinates.

render_bev.

- `recon.render_bev(masks=None,labels=None,ref_frame=0,ego_trajectory=True)`

- *Input:* `masks` is a `PerFrameMask` or an `(N, $N_{obj}$, H, W)` boolean array; `labels` provides per-object names when a raw array is passed; `ref_frame` is the absolute frame index of the reference camera; `ego_trajectory` toggles drawing of the camera path.
- *Output:* a `VisualFeedback` that the agent inspects with `show()`. Stationary objects are annotated with oriented bounding boxes and moving objects with colour-graded trajectory lines.

G.2 `tools.SAM3`

`tools.SAM3` is a client wrapper for the perception serving role and exposes single-frame and video-mode segmentation against SAM3 [3], together with an object-existence check. All segmentation methods return a `PerFrameMask` whose frame indices are absolute video frame indices and align with the indices used by any `Reconstruction` the agent has built.

Image-mode methods. Each operates on a single image and returns a `PerFrameMask` with one frame.

- `segment_image_by_text(image,prompt,label=None)`
  - *Input:* `image` is a PIL image; `prompt` is a free-form text description.
  - *Output:* `PerFrameMask` with one channel per detected instance.
- `segment_image_by_points(image,points,point_labels,label="object")`
  - *Input:* `points` is a list of `[x, y]` pixel coordinates and `point_labels` is a parallel list with values 1 (foreground) or 0 (background).
  - *Output:* `PerFrameMask` for the prompted object.
- `segment_image_by_box(image,box,label="object")`
  - *Input:* `box` is `[x1, y1, x2, y2]` in pixels (xyxy convention).
  - *Output:* `PerFrameMask` for the boxed object.

Video-mode methods. Each propagates the segmentation across the requested frame range, with at most `sam3_max_video_frames` frames per call. `start_frame` and `end_frame` are absolute video frame indices that select a temporal window; `prompt_frame_idx` is local to that window; `video_index` is 1-indexed and selects the source video in multi-video samples.

- `segment_video_by_text(prompts,labels=None,prompt_frame_idx=0,start_frame=None,end_frame=None,video_index=1)`
  - *Input:* `prompts` is a list of text descriptions, one per object.
  - *Output:* `PerFrameMask` over the selected frame range.
- `segment_video_by_points(points_per_object,point_labels_per_object,labels,prompt_frame_idx=0,start_frame=None,end_frame=None,video_index=1)`
  - *Input:* per-object point clicks on the prompt frame, parallel foreground/background labels, and human-readable per-object labels.
  - *Output:* `PerFrameMask` over the selected frame range.
- `segment_video_by_box(boxes,labels,prompt_frame_idx=0,start_frame=None,end_frame=None,video_index=1)`
  - *Input:* per-object bounding boxes on the prompt frame and parallel human-readable labels.
  - *Output:* `PerFrameMask` over the selected frame range.

Object-existence check.

- `is_object_exist(images,object_name)`
  - *Input:* `images` is a list of PIL images and `object_name` is a text description.
  - *Output:* a dictionary with keys `exists` (list of booleans), `counts` (list of instance counts per image), and `summary` (a short text summary).

PerFrameMask object. The returned object exposes `frame_indices`, `labels`, `num_frames`, and `num_objects`, and supports the following access patterns at absolute frame index `fi`.

- `seg.get_mask(frame=fi, object=k)`: 2D boolean mask of object k (`k` may be an index or a string label).
- `seg[fi]`: `(`$N_{\texttt{obj}}$`, H, W)` boolean stack at frame `fi`.
- `seg.get_centroid_3d(recon, frame=fi, object=k)`: median 3D world position of the masked points, or `None`.
- `seg.get_masked_points(recon, frame=fi, object=k)`: `(K, 3)` array of masked world points.
- `seg.visualize(fi)`: `VisualFeedback` overlay for verification with `show()`.

G.3 `tools.Geometry`

`tools.Geometry` is a static class that provides the numerical primitives most commonly needed for spatial reasoning over reconstructed scenes. All methods operate on `numpy` arrays.

Methods.

- `euclidean_distance(p1,p2)`
  - *Input:* two 1D arrays of length 3.
  - *Output:* `float`, the Euclidean distance.
- `angle_between_vectors(v1,v2)`
  - *Input:* two 3D vectors.
  - *Output:* `float`, the angle in degrees.
- `project_point_to_camera(point_3d,c2w,fx,fy,cx,cy)`
  - *Input:* a 3D world point, a 4×4 camera-to-world SE(3) matrix, and the four scalar intrinsics.
  - *Output:* `(u, v)` pixel coordinates, or `None` when the point is behind the camera.
- `rotation_matrix_from_vectors(v_from,v_to)`
  - *Input:* two 3D vectors (need not be unit length).
  - *Output:* a 3×3 rotation matrix that aligns `v_from` with `v_to`.
- `transform_points(points,matrix)`
  - *Input:* a `(..., 3)` array of points and a 4×4 SE(3) matrix.
  - *Output:* a transformed array with the same shape as the input.
- `fit_ground_plane_ransac(points,confidence,conf_threshold=0.3,n_iterations=1000,inlier_threshold=0.05)`
  - *Input:* a `(H, W, 3)` or `(N, 3)` point cloud and a matching confidence array; thresholds and iteration count are optional.
  - *Output:* `(plane_normal, inlier_mask)` via RANSAC [10], or `(None, None)` when fitting fails. The sign of `plane_normal` is unspecified and is disambiguated by the caller.
- `normalized_to_pixel(coords,width,height)`
  - *Input:* a sequence of coordinates in the 0–1000 normalized scale used by `vlm.locate`, together with the image width and height in pixels.
  - *Output:* a `list[float]` of pixel coordinates.

G.4 `tools.Mask`

`tools.Mask` is a static class that provides standard mask statistics over the boolean mask arrays returned by `tools.SAM3`. All inputs are 2D `numpy` arrays with shape `(H, W)` unless otherwise noted.

Methods.

- `centroid(mask)`
  - *Input:* a 2D boolean mask.
  - *Output:* `(cx, cy)` pixel coordinates of the median; `(nan, nan)` when the mask is empty.
- `centroids(masks)`
  - *Input:* an `(N, H, W)` batch of boolean masks.
  - *Output:* a list of `(cx, cy)` centroids, one per element of the batch.
- `area(mask)`
  - *Input:* a 2D boolean mask.
  - *Output:* `int`, the number of `True` pixels.
- `bounding_box(mask)`
  - *Input:* a 2D boolean mask.
  - *Output:* `(x1, y1, x2, y2)` in pixels, or `None` when the mask is empty. For masks with more than 100 `True` pixels, the box is computed from the 1st and 99th percentiles to discard outliers.
- `iou(mask_a, mask_b)`
  - *Input:* two boolean masks of identical shape.
  - *Output:* `float`, the intersection over union.
- `intersection(mask_a, mask_b)`
  - *Input:* two boolean masks of identical shape.
  - *Output:* a boolean mask of the same shape, the element-wise AND.
- `mask_to_bbox(mask)`
  - *Input:* a 2D boolean mask.
  - *Output:* a 4-element `numpy` array `[x1, y1, x2, y2]`, or `None` when the mask is empty.

G.5 `tools.Time`

`tools.Time` is a utility that converts between frame indices and seconds. It is initialized from the input metadata's frame rate and total frame count, and is meaningful only when `Metadata.is_video` is true and `Metadata.fps` is set.

Methods.

- `frame_to_seconds(frame_index)`
  - *Input:* an integer or float frame index.
  - *Output:* `float`, the corresponding time in seconds.
- `seconds_to_frame(seconds)`
  - *Input:* a number of seconds.
  - *Output:* `int`, the nearest frame index, clamped to the valid range.
- `frame_range_to_seconds(start_frame,end_frame)`
  - *Input:* two frame indices.
  - *Output:* `float`, the duration in seconds between them.
- `get_frame_at_time(seconds)`: alias for `seconds_to_frame`.

G.6 `tools.Graph`

`tools.Graph` is a utility that wraps `matplotlib` to produce diagnostic line plots from numerical sequences. The plot is returned as a `VisualFeedback` that the agent inspects inline with `show()` and that also carries a short text summary (minimum, maximum, mean, and trend) accessible as `chart.description`.

Method.

- `plot(values,validity=None,x_label="Frame",y_label="Value",title=None)`
  - *Input:* `values` is a 1D `numpy` array of length T. `validity` is an optional 1D boolean array of the same length whose `False` entries are rendered as gaps. `x_label`, `y_label`, and `title` are display strings.
  - *Output:* a `VisualFeedback` containing the rendered plot and a text description.

G.7 `tools.Draw`

`tools.Draw` is a static class that produces PIL-based annotation overlays. All methods accept a `numpy (H, W, 3) uint8` array, a PIL image, or an `InputImages[i]` entry, and return a PIL image of the same size; the input image is never modified. Colours may be passed as an `(R, G, B) uint8` tuple or as any `matplotlib` or CSS4 colour name. Each method accepts either a single primitive or a list of primitives, and a matching list of colours when per-primitive colouring is desired.

Methods.

- `draw_bbox(image,bboxes,colors=None,thickness=None)`
  - *Input:* `bboxes` is a single `(x1, y1, x2, y2)` in pixels or a list of such boxes.
  - *Output:* a PIL image with axis-aligned rectangle outlines drawn.
- `draw_line(image,lines,colors=None,thickness=None)`
  - *Input:* `lines` is a single `(x1, y1, x2, y2)` in pixels or a list of such segments.
  - *Output:* a PIL image with line segments drawn.
- `draw_point(image,points,colors=None,radius=None)`
  - *Input:* `points` is a single `(x, y)` in pixels or a list of such points.
  - *Output:* a PIL image with filled circles drawn.

H Additional Experiments and Analysis

We provide additional coding-agent comparisons and resource measurements. Each experiment specifies its own backbone and evaluation subset.

H.1 General-Purpose Coding-Agent Comparison

Protocol. We compare SpatialClaw with Claude Code version 2.1.218 using Qwen3.6-27B as the shared backbone and the same serving endpoint. We sample 100 examples per benchmark, except for SpatialTree with 67 examples, using seed 42 across the 20-benchmark suite. Each Claude Code sample runs in a fresh Docker container. Input images are placed in `./frames/` with the same resolution and frame sampling as SpatialClaw. The same tools are available through `import tools`. Tool documentation and reasoning guidance are supplied in `CLAUDE.md`, with adaptations required by the execution environment. Claude Code has access to shell commands, files, and reusable scripts. We evaluate both a 30-step cap and a setting without an imposed step cap; in the capped setting, reminders at steps 28 and 29 request an answer within the remaining budget.

Table 8: Coding-agent comparison with Qwen3.6-27B on the sampled 20-benchmark suite. GPU tool calls count DA3 and SAM3 calls. Token ratios are relative to SpatialClaw.

Metric	No-tool	Claude Code No step cap	Claude Code 30 steps	SpatialClaw 30 steps
Mean accuracy (%)	56.1	62.4	57.3	63.4
Output tokens/sample	6.1K	21.3K	13.5K	8.3K
Output-token ratio	0.73×	2.57×	1.63×	1.00×
GPU tool calls/sample	0	29.4	22.3	5.6
Accuracy − SpatialClaw (pp)	−7.3	−1.0	−6.1	0.0

At the shared 30-step cap, SpatialClaw is 6.1 pp more accurate while using fewer output tokens and GPU tool calls. Removing the coding agent's step cap narrows the accuracy difference to 1.0 pp, with approximately 2.6 times the output tokens and 5.3 times the GPU tool calls of SpatialClaw.

Execution state and interpretation. In SpatialClaw, intermediate masks and reconstructions remain live Python objects across cells. Separate shell processes do not automatically preserve these objects, although a coding agent can serialize them or maintain a long-running interpreter. Observed traces frequently recompute perception outputs in the shell-based setting, consistent with the higher tool-call counts. This comparison evaluates complete execution environments; it is not a single-factor ablation of kernel persistence. It supports an efficiency advantage under the tested protocol, rather than attributing all differences exclusively to persistence.

H.2 Native Coding-Agent Comparison with a Proprietary Backbone

We compare native Claude Code and SpatialClaw using the same Claude Opus 5 backbone, sampling 100 examples per benchmark across 20 benchmarks with seed 42 and the same sampling procedure as the preceding comparison. We evaluate the default 30-step budget and an additional shared 10-step budget. Here, Claude Opus 5 denotes the backbone model; Claude Code denotes the agent environment. The no-tool baseline answers directly with the same backbone.

Table 9: Native Claude Code (CC) and SpatialClaw (SC) with Claude Opus 5. Group scores and the reported overall benchmark mean are shown separately. Differences are SC minus CC in pp.

Benchmark group	No-tool	CC 30 steps	SC 30 steps	Δ	CC 10 steps	SC 10 steps	Δ
Single-image spatial	61.5	71.4	70.9	−0.5	68.1	66.4	−1.7
Multi-view spatial	64.1	78.1	77.9	−0.2	71.8	73.9	+2.1
Video spatial and 4D	52.6	72.6	71.3	−1.3	67.0	67.4	+0.4
General spatial	57.4	74.3	76.9	+2.6	71.6	77.8	+6.2
General video	60.7	70.4	74.4	+4.0	68.1	72.0	+3.9
Mean	58.5	72.9	73.5	+0.6	68.7	70.3	+1.6

At 30 steps, the two environments achieve comparable mean accuracy: 73.5% for SpatialClaw and 72.9% for native Claude Code. Claude Code is stronger in the single-image, multi-view, and video-spatial groups, while SpatialClaw is stronger in general spatial reasoning and general video understanding. Under the 10-step cap, the mean difference grows to 1.6 pp. Both systems substantially improve over the no-tool baseline, by 15.0 and 14.4 pp, respectively, at 30 steps. These comparisons provide evidence for the effectiveness of code-based agents and for the efficiency of SpatialClaw under the tested budgets; the small accuracy differences are not claims of statistical significance.

Table 10: Resource usage with Claude Opus 5 in the default-budget comparison. Reductions are computed from the displayed values relative to native Claude Code.

Metric	Native Claude Code	SpatialClaw	Reduction
Steps/sample	18.9	10.7	43.4%
GPU tool calls/sample	26.3	6.0	77.2%
Peak context consumption	62K tokens	40K tokens	35.5%
LLM cost/sample	$1.19	$0.94	21.0%

At comparable accuracy, SpatialClaw uses fewer interaction steps, perception calls, and context tokens, with lower reported LLM cost. Together with the Qwen3.6-27B comparison, these results suggest that the relative benefits depend on both the backbone and interaction budget. The native comparison measures the full environments rather than isolating any individual implementation choice.

H.3 Computation and Wall-Clock Latency

We report model computation and perception calls separately because a reduction in model turns need not imply a reduction in perception work.

Table 11: Computation across action interfaces. Gemma4-31B, 20 benchmarks, per-question means.

Method	Accuracy (%)	Output tokens	MLLM calls	GPU tool calls
Structured tool-call	56.7	12.7K	14.0	8.3
SpatialClaw	59.9	8.5K	6.7	10.4

SpatialClaw improves accuracy by 3.2 pp while using approximately 33.1% fewer output tokens and 52.1% fewer model calls, with 25.3% more GPU tool calls. The gain therefore does not arise simply from generating more model output, although perception computation increases.

Latency protocol. We compare SpatialClaw with SpaceTools-Toolshed using Gemma4-26B-A4B, processing one sample at a time on identical hardware. This removes concurrent-request scheduling and queuing as confounding factors in the main benchmark setup, where language-model and perception requests use separate servers. Latencies are reported in seconds per sample, with total, LLM, and GPU-tool measurements shown separately. Component medians need not sum to the median total.

Table 12: Controlled sequential latency with Gemma4-26B-A4B. Entries report group latency summaries (seconds); the final row reproduces the reported aggregate. Counts in parentheses denote benchmarks.

Benchmark group	SpaceTools Total	SpaceTools LLM	SpaceTools Tool	SpatialClaw Total	SpatialClaw LLM	SpatialClaw Tool
Single-image spatial (4)	87.5	86.8	0.7	60.0	58.6	1.5
Multi-view spatial (3)	87.1	83.5	2.1	108.1	105.8	1.9
Video spatial and 4D (6)	191.3	166.1	16.0	234.7	154.4	44.3
General spatial (3)	81.1	79.7	0.8	108.0	106.8	2.0
General video (4)	238.3	236.0	1.3	256.8	211.3	11.7
Reported average	137.1	130.4	4.2	153.5	127.4	12.3

The reported aggregate total latency is 153.5 seconds for SpatialClaw versus 137.1 seconds for SpaceTools, an increase of approximately 12.0%. SpatialClaw is faster on single-image tasks and slower in the other four groups.

I Limitations and Broader Impact

I.1 Limitations

The main remaining bottleneck of SpatialClaw is the perceptual quality of the backbone vision-language model and of the perception tools it composes. The failure-mode analysis in the main paper attributes the largest share of remaining errors to perception rather than to the action interface, which means that further interface design has diminishing returns at the current scale of evaluation. We expect the principal axis for future improvement to be perceptual quality, both in the backbone vision-language model and in the underlying reconstruction and segmentation models.

I.2 Broader Impact

SpatialClaw is training-free and adds no parameters to the backbone vision-language model. Existing models can therefore be extended with stronger spatial reasoning ability without additional training data and without fine-tuning, which is particularly valuable in domains where data collection is expensive or impractical, including robotics, embodied applications, and assistive systems. Because the same configuration transfers across backbones and benchmarks without modification, the framework increases the practical value of vision-language models that are already deployed.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state the action-interface study, the SpatialClaw framework, and the evaluation scope (20 benchmarks, eight backbones); these claims are supported by Sec. 4 to 5.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations are discussed in Sec. I.1.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The framework, tools, prompts, baselines, backbones, and evaluation protocol are described in Sec. B to G.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: Code is released at `https://github.com/NVlabs/SpatialClaw`; all benchmarks and backbones are publicly available, and the evaluation protocol and sampling procedure are described in Sec. B.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: SpatialClaw is training-free; the evaluation setup, sample caps, step budget, and backbone configurations are given in Sec. 4 and Sec. B and E.7.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No]

Justification: Results are reported from a single evaluation run per configuration because of the cost of evaluating 20 benchmarks across eight backbones; we do not report error bars, and small differences are not claimed to be statistically significant (Sec. H).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [No]

   Justification: The paper reports per-sample token usage, model and tool call counts, and wall-clock latency (Sec. H.3) and describes the serving setup (Sec. E.1), but it does not report the hardware type or total compute of all experiments.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research uses publicly available benchmarks and models, involves no human subjects, and conforms to the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Positive impacts are discussed in Sec. I.2; the agent executes model-generated code, and the associated risks are mitigated by the sandbox described in Sec. E.4.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper releases no pretrained models or scraped datasets with a high risk of misuse.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All benchmarks, backbone models, and perception tools are cited at their first use (Sec. 4, Sec. E.7 and G).

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.

- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The released code is documented in its repository at `https://github.com/NVlabs/SpatialClaw`.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: VLM backbones are the core reasoning component of SpatialClaw (Sec. 3), and an LLM judge (Gemini-3.1-Pro) is used for the win-attribution and failure-mode analyses (Fig. 7); both are described in the paper.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.