TRANSDUCTIVE VISUAL PROGRAMMING: EVOLVING TOOL LIBRARIES FROM EXPERIENCE FOR SPATIAL REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

The composition of specialized tools offers a powerful approach for complex visual reasoning, particularly for tasks involving 3D spatial understanding. However, existing visual programming methods are often constrained by fixed toolsets or offline tool induction, which leads to suboptimal solutions and poor tool reuse. We introduce **T**ransductive **V**isual **P**rogramming (TVP), a novel framework that dynamically evolves a library of reusable tools by learning from its problem-solving experience. TVP abstracts recurring solution patterns into new, higher-level tools, which are then used to construct simpler and more effective programs for new tasks. On the challenging Omni3D-Bench, TVP establishes a new state of the art, outperforming both specialized vision-language models and prior visual programming systems. The evolved tools also exhibit strong generalization to out-of-domain queries on 3DSRBench, SpatialSense, and VGBench. Our work demonstrates that transductive tool evolution is a powerful and generalizable paradigm for building robust visual reasoning systems.

1 INTRODUCTION

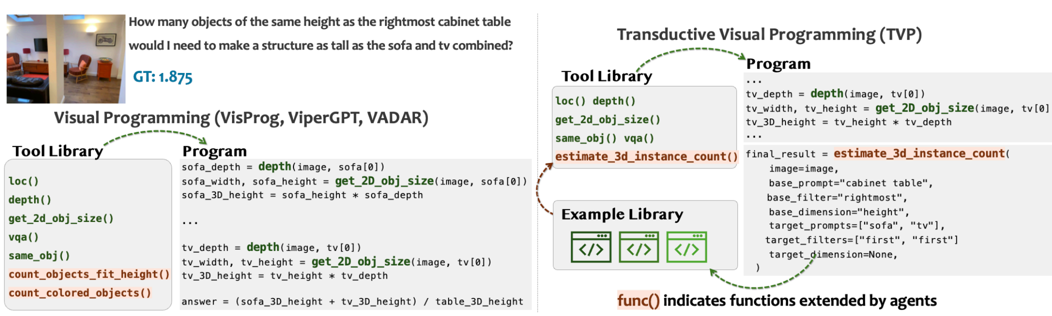

Figure 1: (Left) Prior methods operate in an open-loop manner—tools are created without experience from actual problem-solving. (Right) TVP maintains both an Example Library of successful program solutions and a Tool Library of abstracted functions. Through this closed-loop system, TVP abstracts tools from patterns in proven solutions.

Reasoning on complex visual scenes, including localization, understanding spatial relations, and counting objects, is not a single visual skill. While pre-trained Vision-Language Model (VLM) have made progress in processing images, spatial reasoning remains challenging for frontier models such as GPT-4o (Lee et al., 2025; Marsili et al., 2025). This limitation motivates a compositional approach: decomposing complex visual problems into discrete computational steps executable by specialized tools — a paradigm known as visual programming.

An effective programming system should emulate how human programmers learn: first by solving concrete problems, and only then by abstracting recurring patterns into reusable functions. Current visual programming paradigms do the opposite. They either rely on fixed, predefined tool sets that cannot learn (Gupta & Kembhavi, 2023; Surís et al., 2023), or they speculatively synthesize new tools inductively before their utility is proven. For example, VADAR (Marsili et al., 2025), a prior system that induces tools from questions upfront without grounding to any solutions, still relies 94.2% of the time on the basic predefined APIs despite the presence of its proposed tools.

We propose **T**ransductive **V**isual **P**rogramming (TVP), a framework that generates program solutions while continually evolving a dynamic and highly-reusable tool library built from past experience. Analogous to human programmers who identify recurring patterns and abstract them into reusable functions only after solving concrete problems, TVP learns through *transductive* abstraction. This approach *directly leverages successful program solutions from specific examples to form new, higher-level tools*. Rather than "inducing" potentially useful functions before solving problems, TVP recognizes repeated patterns after implementing multiple concrete solutions, ensuring that learned abstractions are useful and grounded in experience.

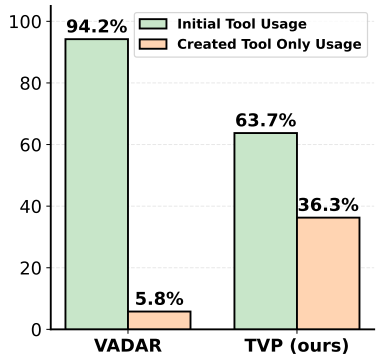

Figure 2: Tool usage distribution: transductive (TVP) vs. inductive abstraction (VADAR)

As illustrated in Fig. 1, TVP's architecture is centered around a dual-library system designed to facilitate this transductive learning loop. The **Example Library** serves as a memory, accumulating a growing corpus of successful program solutions. In parallel, the **Tool Library** maintains a dynamic set of higher-level functions that are continuously evolved by abstracting common patterns from the Example Library. When faced with a new query, TVP first retrieves relevant demonstrations from its experience, then leverages its evolved, higher-level tools to construct simpler and more efficient programs.

On the challenging Omni3D-Bench for 3D spatial reasoning, TVP achieves a new state-of-the-art overall accuracy of 33.3%, outperforming both generic VLMs like GPT-4o (+22.4%) or SpaceMantis (Jiang et al., 2024; Chen et al., 2024) built with spatial-specific finetuning, and previous visual programming systems including VADAR (+11.3%). The superior performance of TVP is a direct result of its effective tool discovery and reuse. Our analysis shows that tools learned through TVP are used far more frequently than those of baseline methods; 36.3% of our learned tools are employed as the only functions in final program solutions, compared to just 5.8% for VADAR's speculatively induced APIs.

Moreover, the tool set evolved by TVP generalizes to unseen tasks with strong performance. After the TVP agent evolves its dual libraries on Omni3D-Bench §3.1, it is directly applied to handle novel spatial reasoning queries sampled from SpatialScore-*Hard* collection §3.2 (including 3DSR-Bench (Ma et al., 2024), SpatialSense (Yang et al., 2019), and VG-Bench (Wu et al., 2025)), where it also delivers superior performance with zero-shot generation across task categories, demonstrating that its learned skills are highly transferable. Taken together, this paper demonstrates that programming with transductive tool creation from experience is a powerful paradigm for tackling complex spatial reasoning tasks.

2 Transductive Visual Programming

Transductive Visual Programming (TVP) is a framework that learns to create and refine a library of reusable tools from its own problem-solving experience. It operates via a closed-loop process centered on a dual-library architecture (Fig. 3): an **Example Library** $\mathcal{E}$ accumulates successful program solutions, while a **Tool Library** $\mathcal{T}$ maintains an evolving set of functions abstracted from the experience. This design allows TVP to emulate human learning: first solve concrete problems, then generalize successful patterns into reusable skills. The entire workflow is formalized in Alg. 1.

2.1 Program Generation and Execution

Initialization. TVP begins with an empty Example Library $\mathcal{E} \leftarrow \emptyset$ and a Tool Library $\mathcal{T}$ initialized with predefined basic vision tools. These predefined tools, inherited from Marsili et al. (2025), include: object localization (`loc`) and bounding box detection (`get_2d_object_size`)

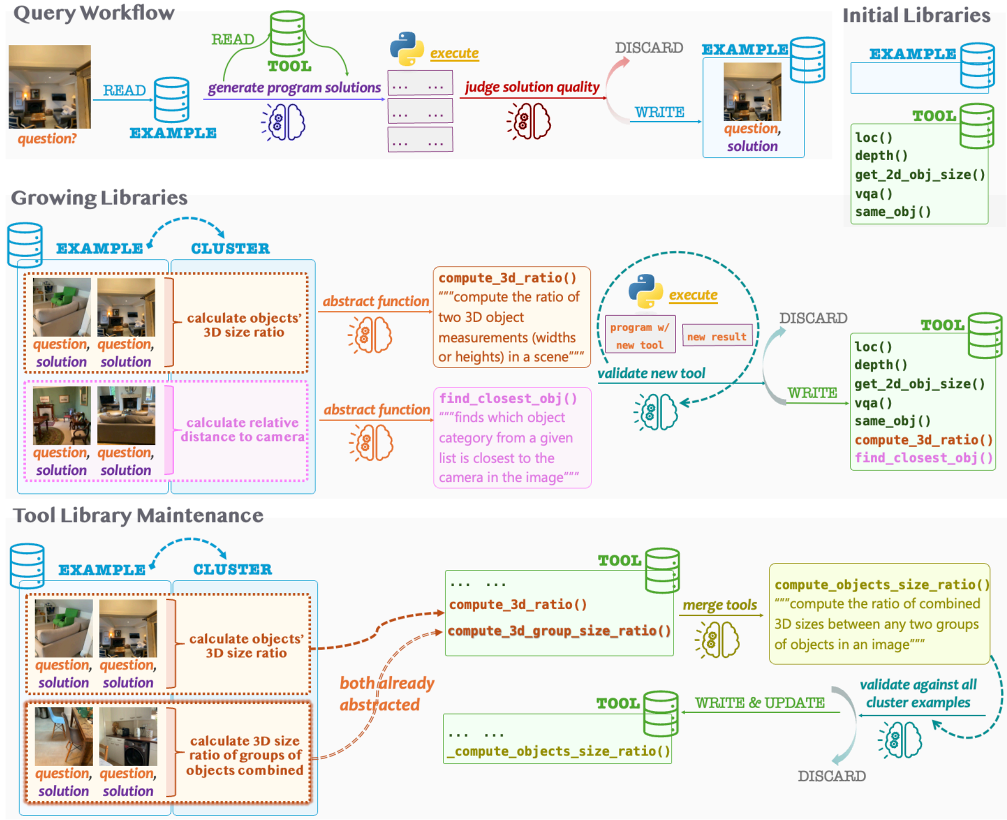

Figure 3: **TVP's dual-library architecture and pipeline.** (Top) Query workflow: For each visual reasoning question, TVP retrieves similar examples from the Example Library, generates candidate programs using available tools and high-quality solutions join the Example Library. (Middle) Tool abstraction: As examples accumulate, TVP clusters similar queries and abstracts common solution patterns into parameterized tools. (Bottom) Tool maintenance: When functionally similar tools emerge from different clusters, TVP identifies and merges them into unified tools that generalize both functionalities.

with GroundingDINO (Liu et al., 2024), depth estimation (`depth`) via UniDepth (Piccinelli et al., 2024), object property queries with GPT-4o (`vqa`), and overlapping bounding box verification with `same_object`. These basic spatial computations serve as building blocks for more complex reasoning.

Example retrieval. Given a question q_i, TVP first retrieves k most similar queries from the current Example Library $\mathcal{E}$. Retrieval is based on the embedding similarity of question texts. Since question texts are straightforward, similar embeddings indicate similar query semantics.

Program generation and execution. We provide the question q_i, the retrieved k examples with their solutions as in-context demonstrations, and the current Tool Library $\mathcal{T}$ (in the form of tool signatures and docstrings) to the Program Generator, which is an LLM that explores m different candidate program solutions for q_i, as visual reasoning problems often admit multiple valid approaches. TVP executes each program candidate with access to the full Tool Library $\mathcal{T}$ implementations, producing execution traces as namespaces and a final calculated answer to q_i.

2.2 Example Library: Adding High-Quality Programs

Quality judge. TVP employs a VLM judge to assess each candidate program's quality. The judge has access to the program implementation, full execution trace and the produced answer, evaluating them against the question and image as visual evidence. The best candidate represents the final solution to the question. More details on the judge criteria are provided in §C.3.

Joining Example Library. We maintain a quality threshold τ_q that gates entry to the Example Library. This ensures $\mathcal{E}$ contains only high-quality program solutions, forming the foundation for both effective in-context examples (§2.1) and high-quality tool abstractions (§2.3).

2.3 TOOL LIBRARY: TOOL ABSTRACTION FROM EXPERIENCE

At intervals of n_a processed questions, TVP mines its Example Library for recurring patterns that merit abstraction into reusable tools for the Tool Library (see Alg. 2 2 for the complete abstraction process).

Example clustering. First, all queries in $\mathcal{E}$ are clustered by embedding similarity of question texts (similar to example retrieval). This creates initial groups of related questions with high potential for similar program solutions. For clusters with similarity surpassing threshold τ_{sim} and size exceeding τ_{cluster}, we query an LLM for abstraction analysis to evaluate the cluster's abstraction potential. High-potential patterns (score $\geq \tau_{\text{potential}}$) trigger tool abstraction from these examples' solutions. More details on the criteria for abstraction potential are provided in §C.3.

Tool abstraction. We provide the Tool Abstractor with all cluster examples' questions, program solutions, execution results, and the current Tool Library. The Tool Abstractor creates a parameterized function capturing the cluster's shared program logic. For instance, as shown in Fig. 3, clusters calculating 3D size ratios yield `compute_3d_ratio`, while clusters finding nearest objects produce `find_closest_obj`. The new function replaces step-by-step programming in the cluster examples.

Validating new tool. Each new abstract function is rigorously vetted (Alg. 3). TVP rewrites each cluster example using the new tool; with no ground truth, a rewrite is accepted if it yields identical results or—common for floating-point cases—is judged by a VLM to be equally valid or better given the visual evidence. This safeguards or improves visual-program quality while enabling generalization. Unlike prior speculative induction, TVP is transductive: it lifts concrete solutions built from low-level tools directly into abstractions, ensuring every new tool is grounded in experience from the Example Library.

Example status update. When cluster examples are abstracted into a common function, we rewrite these examples in the Example Library to use the new tool. This better guides future similar questions to employ abstract tools in their solutions. Once successfully abstracted, we mark these cluster examples as closed for future clustering and abstractions.

2.4 DUAL-LIBRARY MAINTENANCE

As more datapoints are processed and both libraries grow, TVP performs periodic maintenance for both libraries.

Periodic Tool Library update with tool merging. Tool abstractions performed with accumulative cluster examples may lead to functionally similar abstract tools created at different intervals. We periodically merge functionally similar tools in $\mathcal{T}$ (see Alg. 4). As illustrated in Fig. 3, tools like `compute_3d_ratio` and `compute_3d_group_size_ratio` serving similar purposes merge into a more general `compute_objects_size_ratio`. Merged tools undergo the same rigorous validation against all examples using the original tools. This periodic merging reduces redundancy while preserving all abstracted functionalities.

Example Library update across iterations. TVP runs through multiple iterations T over the entire dataset $\mathcal{D}$ (as shown in Alg. 1). Examples in the Example Library can be updated across iterations by comparing quality ratings—better program solutions replace existing ones, as later iterations with access to more abstract tools produce more efficient and cleaner programs.

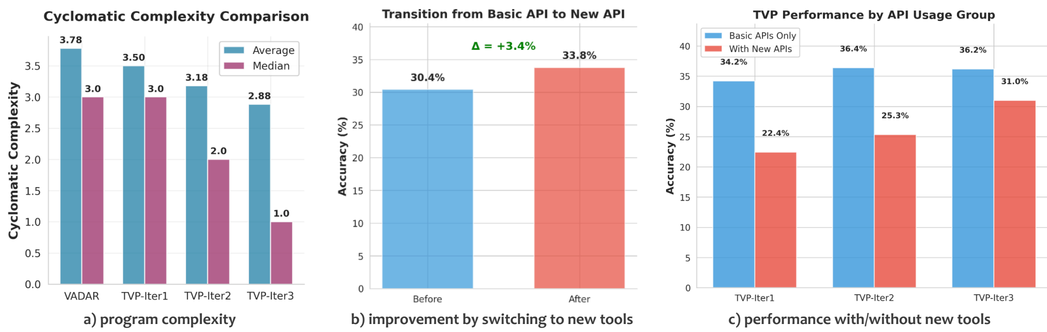

Figure 4: Tool abstraction improves both program efficiency and accuracy. (a) Program cyclomatic complexity steadily decreases across TVP iterations. (b) Programs gain $+3.4\%$ accuracy when switching from basic to abstracted tools. (c) Performance with new APIs improves 38% across iterations as TVP gradually masters its learned abstractions

3 Experiments

3.1 3D Spatial Reasoning Performance

TVP is a general visual programming system applicable to various tasks given appropriate initialization (particularly the initial toolkit, §2.1). 3D spatial reasoning is a representative challenge where mere visual perception in the form of VQA fails short of the precise geometric calculations required by the 3D spatial inference.

Setup. We evaluate our approach on **Omni3D-Bench** (Marsili et al., 2025), a challenging test set containing 501 non-templated spatial queries on diverse real-world scenes, requiring 3D object grounding and reasoning about distances and dimensions. Among the four question types, we evaluate: (1-3) Yes/No, Multiple Choice, and Counting via exact-match accuracy (with possible fuzzy string normalization); (4) Float-point calculations via Mean Relative Accuracy (Yang et al., 2025): $\mathcal{MRA} = \frac{1}{|C|}\sum_{\theta \in C} \mathbb{1}\left(\frac{|\hat{y}-y|}{y} < 1-\theta\right)$ with $C = \{0.5, 0.55, ..., 0.95\}$, and Float($\pm 10\%$) accuracy for predictions within 10% error tolerance. We run TVP for $T = 3$ iterations with quality threshold $\tau_q = 8.5$ for example library entry, similarity threshold $\tau_{\text{sim}} = 0.8$, and minimum cluster size $\tau_{\text{cluster}} = 4$ for clustering. Tool abstraction and deduplication occur at every step ($n_a = n_d = 1$). We use GPT-4o for program generation and image-based quality judge, and 4o-mini for abstraction and auxiliary tasks.

For baselines, we compare against generic VLMs (GPT-4o, LLaVA-OneVision-7B-Chat Li et al., 2024, Qwen2-VL-7B-Inst Wang et al., 2024a, Molmo-7B-D Deitke et al., 2025), spatial-finetuned VLMs (SpaceMantis Jiang et al., 2024[1] and SpatialBot-3B Cai et al., 2025), and prior visual programming systems (VisProg Gupta & Kembhavi, 2023, ViperGPT Surís et al., 2023, VADAR Marsili et al., 2025). Refer to §C for additional implementation details.

TVP achieves state-of-the-art through experience-grounded tool creation. As shown in Tab. 1, TVP achieves new state-of-the-art performance on Omni3D-Bench with 33.3% overall accuracy, outperforming all baselines including the previous best visual programming system VADAR (29.9%) by 11.4% relative improvement. The advantage is most pronounced on precise floating-point spatial calculations: TVP reaches 19.3% accuracy within ±10% tolerance, vs. VADAR's 15.9% and GPT-4o's 8.2%. Monolithic VLMs handle perception-heavy yes/no and multiple-choice tasks reasonally well, but falter on exact 3D measurements—where compositional programming excels. Even spatial-finetuned models (*e.g.*, SpaceMantis) show no meaningful gains over generic VLMs on these calculations, again underscoring the necessity for stronger compositional programming approach like TVP.

TVP enables effective program compression. As TVP creates more higher-level tools, repetitive low-level operations are gradually eliminated. Fig. 4 (panel a) illustrates how our TVP steadily re-

[1]Finetuned following SpatialVLM (Chen et al., 2024)

Table 1: Performance comparison on Omni3D-Bench. **Best scores bolded**, second best underlined. *Results from VADAR paper.

Method	Accuracy by Question Type (%)					Overall (%)
	Yes/No	Multiple Choice	Counting	Float MRA	Float (±10%)	
Generic VLMs						
GPT-4o	**65.3**	60.5	18.6	26.7	8.2	27.2
Qwen2-VL-7B-Inst	58.7	33.7	12.9	21.5	10.0	21.8
LLaVA-OV-7B-Chat	60.0	27.9	22.9	26.8	11.1	23.0
Molmo-7B-D	46.7	41.9	18.6	28.4	8.9	21.6
Spatial-Finetuned VLMs						
SpaceMantis	53.3	30.2	4.3	21.4	8.2	18.2
SpatialBot-3B	60.0	30.2	0.0	17.7	8.5	18.8
Visual Programming						
VisProg*	54.7	25.9	2.9	0.9	–	–
ViperGPT*	56.0	42.4	20.0	15.4	–	–
VADAR	56.0*	57.6*	21.7*	35.5*	15.9	29.9
TVP (ours)-iter1	50.7	**62.8**	21.4	34.7	18.5	31.3
TVP (ours)-iter2	60.0	59.3	**24.3**	34.7	17.4	31.9
TVP (ours)-iter3	60.0	61.6	**24.3**	**36.5**	**19.3**	**33.3**
w/o Tool Lib	60.0	61.6	21.4	35.5	17.0	31.7

duce program complexity across iterations. The average McCabe's Cyclomatic Complexity Number (CCN) (McCabe, 1976) (§C.2) decreases from 3.5 to 2.88 (-17.7%), with the median CCN dropping from 3.0 to 1.0. The program compression leads to two key benefits: (1) improved efficiency and reduced potential errors in reimplementation of branching logic; (2) improved interpretability of generated programs, as a single function call replace otherwise whole paragraphs of code (see qualitative examples in Fig. 10).

Tool abstraction facilitates progressive self-improvement, especially on hard problems. To isolate the contribution of our tool abstraction from in-context learning with examples, we run TVP with only the Example Library active (*w/o Tool Lib* in Tab. 1). This configuration still benefits from retrieved similar examples as few-shot demonstrations but relies solely on basic initial tools. Notably, the Example-Library-Only variant **already achieves a competitive** 31.5% **accuracy overall, outperforming all prior baselines**. This strong performance demonstrates the quality of our accumulated experience, enabled by our example library admission design (§2.2).

While the Example Library provides strong foundation, our full TVP system with active tool creation shows more **significant improvement across iterations** (31.3% → 31.9% → 33.3%), while the Example-Library-Only variant maintains static (31.7% → 31.5% → 31.5%). This progressive improvement stems from the **closed-loop design** as illustrated in Fig. 1: abstracted tools encapsulate past experience and enable better future programs, which become better examples, from which better tools can be abstracted. Fig. 4 (panel b) also indicates this effect: **when programs switch from basic tools to newly created abstractions, they achieve** +3.4**% accuracy improvement**. Without tool creation in the loop, this self-improving cycle is weakened.

Furthermore, we find that the tool abstraction **contributes most value on hard problems**. We use GPT-5 with high reasoning effort to rate the difficulty of the spatial reasoning questions on a 1.0–10.0 scale (details in §C.4), then divide questions into three groups: Easy (1–3), Medium (4–6), and Hard (7–10). Fig. 5 shows accuracy across methods for different complexity levels. **TVP (Full) delivers the best performance on both Easy and Hard batches**. For easy questions, thoroughly validated created tools avoid potential reimplementation errors, leading to more stable performance. For harder questions, created tools provide simpler solution steps that eliminate complicated logic, thus easing the program reasoning. Fig. 6 reveals the evolution of the benefits brought by our active tool abstraction across iterations, as we compare the the performance delta between TVP (Full) and TVP (Example-Lib-Only) for each complexity level. **On the hardest batch, TVP (Full) shows the most significant improvement trajectory**, starting at −4.5% relative to Example-Lib-

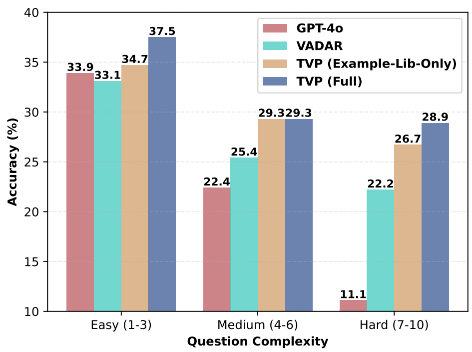

Figure 5: Performance comparison across question complexity levels on Omni3D-Bench. The complexity scores are rated with criteria defined in §C.4

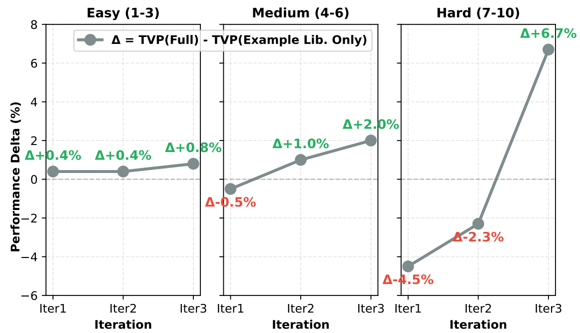

Figure 6: Performance delta between TVP (Full) and TVP (Example-Lib-Only) across iterations for each question complexity level. TVP full system (with active tool creation) shows most significant gains on the hardest batch of questions.

Only in iteration 1, but ultimately surpassing it by +6.7% in iteration 3. This demonstrates that our created tools – beyond just in-context examples – effectively reduce the reasoning workload for most challenging questions, as they encapsulate past experience of complicated code logic into simple function calls. An example is illustrated in Fig. 10(panel b), where a newly created tool serves as a convenient step in solving a spatial reasoning problem.

Both libraries evolve steadily through transductive learning. Fig. 9 visualizes TVP's evolution through three iterations on Omni3D-Bench. The system exhibits steady growth in both libraries: the Example Library accumulates from 0 to 304 high-quality solutions while the Tool Library expands from 5 initial tools to 11 active abstractions (after creating 61 total and merging redundant ones). This controlled growth—creating many abstract functions but keeping only the most general through periodic merging (§2.4)—ensures the tool library remains manageable while capturing diverse functionalities. The impact of this evolution is evident in Fig. 4(panel c). Programs using only basic APIs maintain stable performance across iterations, confirming that TVP preserves its ability to leverage fundamental tools. Meanwhile, programs utilizing new APIs show dramatic accuracy improvement—from 22.4% in iteration 1 to 31.0% in iteration 3—as the system masters applying its learned abstractions. This +38% relative improvement demonstrates that TVP doesn't just create tools but learns to use them more effectively over time.

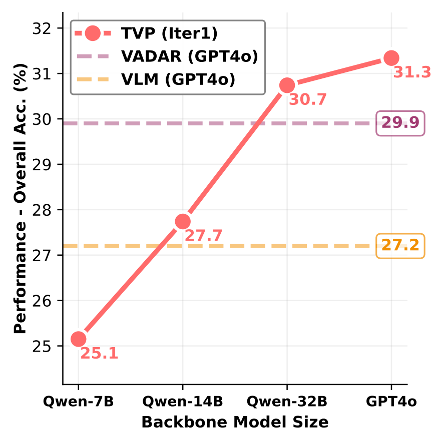

Figure 7: TVP performance scales consistently on Omni3D-Bench with backbone model capacity (Qwen2.5-Coder-7/14/32B & GPT-4o).

TVP is robust to backbone LLM choice, showing a clear scaling trend with model sizes. We further investigate TVP's robustness to open-source smaller LLMs, represented by the Qwen2.5-Coder-Instruct (Hui et al., 2024) family as the backbone program generator, spanning from 7B to 32B parameters. Specific configurations are given in §C.2. Fig. 7 presents the scaling behavior, where TVP exhibits a clear performance improvement with increasing model capacity. Notably, using an open-source 32B model, TVP achieves performance close to our GPT-4o-backed variant (30.7% vs. 31.3%), and surpasses the previous best baseline VADAR (29.9%) despite its more capable GPT-4o backbone. This result underscores that our TVP **does not rely on proprietary-specific optimal LLMs but can achieve strong performance with more accessible open-source alternatives**. The consistent scaling trend also validates TVP's architecture as model-agnostic, and suggests significant **future potential** of our transductive tool creation, as foundation models with enhanced capabilities become available.

3.2 Generalizing Across Unseen Spatial Reasoning Queries

As TVP builds its dual libraries through one test set (Omni3D-Bench), we evaluate whether the learned capabilities transfer to unseen spatial reasoning queries in the wild. We take the agent built from §3.1 and directly apply it to new benchmarks without any additional example or tool creation.

Setup. We sample from the SpatialScore-*Hard* collection (Wu et al., 2025) that contains the most challenging spatial understanding queries. In total, we collected 256 test datapoints from 3DSR-Bench (Ma et al., 2024), SpatialSense (Yang et al., 2019), and VG-Bench (Wu et al., 2025) respectively, keeping single-image, visual-bounding-box-free samples (aligning with the structural setup on Omni3D-Bench). These encompass 4 categories of spatial reasoning capabilities (Fig. 8), covering Yes/No, Multiple-Choice, and Open-Ended (numeric calculation) questions types. Same as in §3.1, we evaluate the former two question types with accuracy, and the last type with Float($\pm 10\%$) accuracy within 10% error range tolerance. We compare our TVP agent's zero-shot transfer performance against various VLMs and VADAR as representative visual programming system. Notably, we run VADAR on these test sets, meaning that it has new tools created specifically for these test sets while our TVP directly applies its Omni3D-Bench libraries without any modification.

Table 2: Results on benchmarks from sampled SpatialScore-*Hard* collection; TVP (built on Omni3D-Bench) evaluated zero-shot. **Best scores bolded**, second underlined.

Method	3DSR-B.	SpatialSense	VG-B.	Overall
Generic VLMs				
GPT-4o	52.1	46.5	20.3	42.6
LLaVA-OV-7B-Chat	12.4	9.9	9.4	10.9
Qwen2-VL-Inst	49.6	32.4	7.8	34.4
Molmo-7B-D	41.3	54.9	12.5	37.9
Spatial-Finetuned VLMs				
SpaceMantis	37.2	19.7	7.8	25.0
SpatialBot-3B	20.7	**62.0**	6.2	28.5
Visual Programming				
VADAR	24.8	40.8	39.1	32.8
TVP$_{\text{Generalizing}}$	**52.9**	59.2	**43.8**	**52.3**

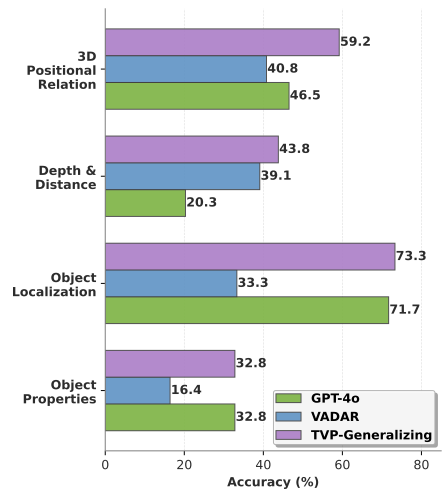

Figure 8: TVP's libraries transfer well on SpatialScore-*Hard*, particularly for 3D spatial and depth/distance reasoning.

Transductively learned tools generalize with superior performance. As shown in Tab. 2, TVP achieves 52.3% overall accuracy on SpatialScore-*Hard*, outperforming even VADAR (32.8%) which inductively creates tools specifically for these test sets. The performance breakdown in Fig. 8 shows TVP excels particularly on challenging spatial reasoning categories. For 3D positional relations, TVP achieves 59.2% vs VADAR's 40.8% and GPT-4o's 46.5%. On depth and distance estimation (from VG-Bench), TVP reaches 43.8% while VADAR manages 39.1% and GPT-4o only 20.3%. The consistent superior performance across spatial tasks validates that our transductive learning builds up genuinely reusable libraries rather than overfitted solutions. Fig. 10 (panel b) provides qualitative evidence of this generalization. Tools like `find_largest_by_3d_metric`, originally abstracted from Omni3D-Bench solutions, now handles the a new 3D comparison queries from 3DSR-Bench without any modification, demonstrating that transductive abstraction captures fundamental reasoning patterns rather than dataset-specific tricks.

3.3 Qualitative Analysis On Tool Utilization And Evolution

Transductively abstracted tools are easily applicable to diverse tasks. Fig. 10 illustrates how TVP's tool creation achieves high utilization. Panel (a) shows `estimate_3d_instance_count` handling diverse ratio calculations within Omni3D-Bench—from computing how many cabinet tables would match a sofa-TV height to determining television stacking requirements. Panel

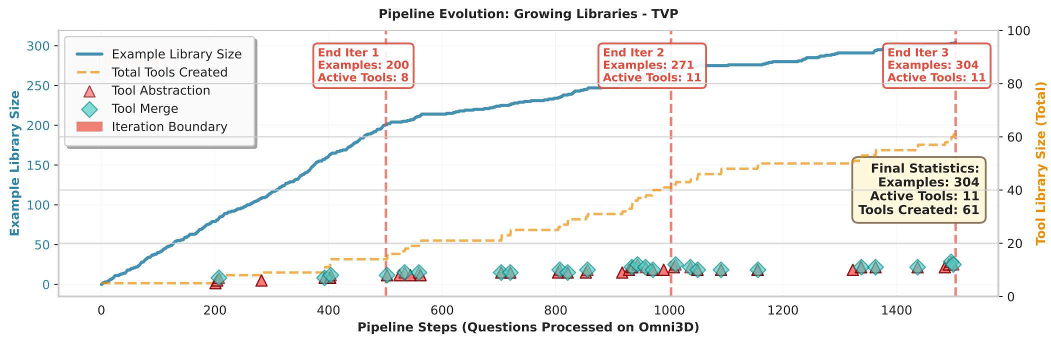

Figure 9: Evolution of TVP's dual libraries through iterations on Omni3D-Bench. The Example Library grows steadily to 304 solutions while the Tool Library expands strategically—creating 61 tools total but maintaining 11 active abstractions with periodic merging.

(b) reveals even stronger evidence: `find_largest_by_3d_metric` seamlessly transfers from Omni3D-Bench (comparing armchair and fireplace heights) to completely unseen 3DSR-Bench queries (comparing train and street light elevations). These tools succeed because they encode proven solution patterns validated through actual problem-solving, not hypothetical utilities.

Tool hierarchies emerge naturally through iterative refinement. Fig. 11 traces the evolution of a representative tool hierarchy. Starting from basic step-by-step solutions using primitive operations (Level-0), TVP first abstracts `estimate_3d_height_from_reference` to handle height estimation tasks (Level-1). As more examples accumulate, the tool maintenance mechanism identifies similarity with a width-estimation function, merging them into the more general `estimate_object_dimension_by_reference` (Level-2). Later iterations add filtering capabilities, creating an even more powerful abstraction. This hierarchical evolution mirrors how human programmers refactor code—starting with specific solutions, recognizing patterns, and progressively generalizing. The Tool Library maintenance mechanism (Fig. 3) ensures this evolution produces increasingly powerful tools while avoiding redundancy. Through periodic merging, functionally similar tools like `compute_3d_ratio` and `compute_3d_group_size_ratio` combine into unified abstractions that handle broader use cases. The quantitative impact is clear: programs using these evolved tools achieve both higher accuracy (+3.4% when switching to new tools, Fig. 4-panel b) and lower complexity (median cyclomatic complexity drops 66% from 3.0 to 1.0, Fig. 4-panel a). This demonstrates that transductive visual programming doesn't just solve problems—it continuously improves its problem-solving capabilities through experience.

4 Related Work

4.1 Spatial Reasoning

Spatial reasoning requires understanding of precise real-world spatial relationships between 3D objects beyond the pixel space of images, which is challenging for monolithic VLMs (Kamath et al., 2023; Majumdar et al., 2024; Fu et al., 2024; Tong et al., 2024; Cai et al., 2025; Zhang et al., 2024). Even for recent VLMs built with specialized spatial finetuning (*e.g.*, SpatialVLM Chen et al., 2024, SpatialRGPT Cheng et al., 2024, SpatialBot Cai et al., 2025), these models still struggle with more diverse 3D reasoning queries (Lee et al., 2025; Marsili et al., 2025) – similar to our findings revealed from experimenting on SpatialVLM and SpatialBot (§3.1). These limitations motivated visual programming approaches that decompose complex visual tasks into executable steps, where each step leverages specialized vision tools. VisProg (Gupta & Kembhavi, 2023) introduced a domain-specific language (DSL) for composing vision specialists. ViperGPT (Surís et al., 2023) directly generates Python code for calling APIs. Both systems rely on predefined static APIs, limiting their ability to handle queries beyond their initial toolkit. VADAR (Marsili et al., 2025) attempts a dynamic tool set by **proposing** potentially useful functions as new APIs, which are created through pure **"induction"**—based solely on question texts before solving any problems. This speculative approach leads to low utilization of the generated APIs in actual solutions (see Fig. 2). Our TVP belongs to the

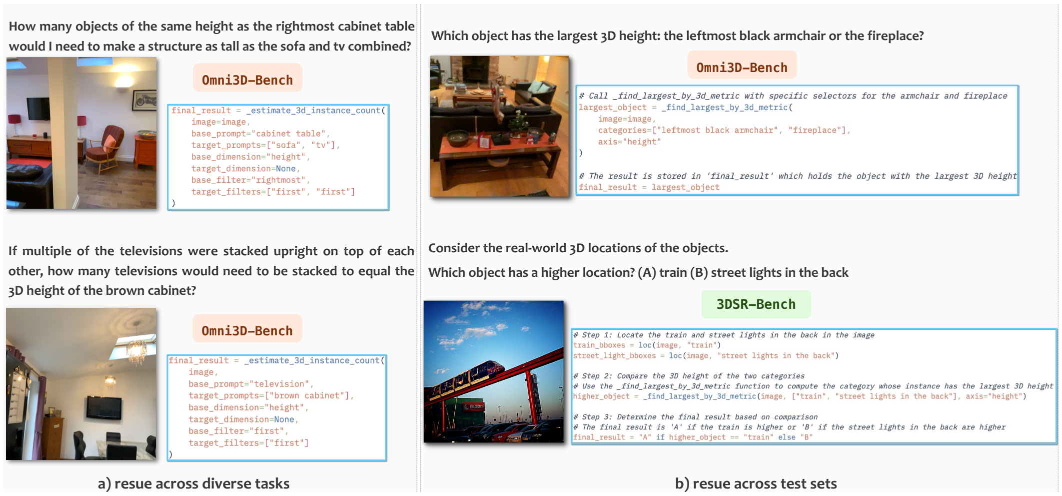

Figure 10: Tool reuse across diverse tasks and benchmarks. (a) Transductively learned tools handle diverse problems within Omni3D-Bench. (b) The tool learned from Omni3D-Bench transfers to unseen benchmarks.

visual programming family but takes a fundamentally different approach: rather than inductively proposing tools, it learns them **"transductively"** through experience—solving problems with basic tools first, then abstracting recurring solution patterns into new functions.

4.2 Tool Use and Abstraction

Agentic systems calling various specialist tools have shown superior performance across a wide range of domains, including web navigation (Zheng et al., 2025; Wang et al., 2025), robotic controls (Liang et al., 2022), graphics generation (Hu et al., 2024; Sun et al., 2025) and game exploration (Wang et al., 2023). In 3D visual tasks, Yuan et al. (2024) proposes view-dependent and -independent modules in visual programs for zero-shot open-vocabulary grounding. Mi et al. (2025) introduces "code as spatial relation encoders" optimized through test suites before deployment. Beyond applying specialized tools, recent works have studied the automatic creation of new tools that enable self-evolving agents with a dynamic toolbox (Cai et al., 2023; Wang et al., 2024b; Yuan et al., 2023; Qian et al., 2023). LILO (Grand et al., 2023) compresses programs into symbolic λ-expressions for abstracting tools. Alita (Qiu et al., 2025) produces specialized model context protocols (MCP) connecting web-search with tool generation and execution. Skillweaver (Zheng et al., 2025) identifies novel skills from web tasks following a simple-to-complex curriculum. ASI (Wang et al., 2025) shares our insight that tool abstractions come from experience in concrete solutions, and proposes novel functions from concrete action trajectories in web environments. Our TVP evolves its toolbox through a unique dual-library architecture that simultaneously gathers experience and creates tools. By grounding tool abstraction in actual problem-solving experience, TVP demonstrates how transductive library learning produces effective self-evolving visual programming agent. §B provides an extended comparison between TVP and prior work on tool discovery.

5 Conclusion

We introduce Transductive Visual Programming (TVP), a novel paradigm where agents learn to build tools from problem-solving experience, moving beyond static or speculatively-created tool sets. By abstracting reusable functions from successful solutions, TVP mirrors human tool learning process. Our approach sets a new state-of-the-art on challenging 3D spatial reasoning benchmark Omni3D-Bench. The learned tools also demonstrate strong zero-shot generalization to unseen spatial queries from other test sets (3DSRBench, SpatialSense, and VGBench), proving that TVP builds robust, reusable knowledge. Our work opens exciting directions: TVP's dual-library architecture is task-agnostic, and demonstrates a viable path toward continually self-learning agents that build hierachical, compositional skills.

REFERENCES

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023.

Wenxiao Cai, Iaroslav Ponomarenko, Jianhao Yuan, Xiaoqi Li, Wankou Yang, Hao Dong, and Bo Zhao. Spatialbot: Precise spatial understanding with vision language models. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9490–9498. IEEE, 2025.

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 *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14455–14465, 2024.

An-Chieh Cheng, Hongxu Yin, Yang Fu, Qiushan Guo, Ruihan Yang, Jan Kautz, Xiaolong Wang, and Sifei Liu. Spatialrgpt: Grounded spatial reasoning in vision-language models. *Advances in Neural Information Processing Systems*, 37:135062–135093, 2024.

Matt Deitke, Christopher Clark, Sangho Lee, Rohun Tripathi, Yue Yang, Jae Sung Park, Mohammadreza Salehi, Niklas Muennighoff, Kyle Lo, Luca Soldaini, et al. Molmo and pixmo: Open weights and open data for state-of-the-art vision-language models. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 91–104, 2025.

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*, pp. 148–166. Springer, 2024.

Gabriel Grand, Lionel Wong, Maddy Bowers, Theo X Olausson, Muxin Liu, Joshua B Tenenbaum, and Jacob Andreas. Lilo: Learning interpretable libraries by compressing and documenting code. *arXiv preprint arXiv:2310.19791*, 2023.

Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 14953–14962, 2023.

Ziniu Hu, Ahmet Iscen, Aashi Jain, Thomas Kipf, Yisong Yue, David A Ross, Cordelia Schmid, and Alireza Fathi. Scenecraft: An llm agent for synthesizing 3d scenes as blender code. In *Forty-first International Conference on Machine Learning*, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Dongfu Jiang, Xuan He, Huaye Zeng, Con Wei, Max Ku, Qian Liu, and Wenhu Chen. Mantis: Interleaved multi-image instruction tuning. *arXiv preprint arXiv:2405.01483*, 2024.

Danial Kamali and Parisa Kordjamshidi. Neptune: A neuro-pythonic framework for tunable compositional reasoning on vision-language. *arXiv preprint arXiv:2509.25757*, 2025.

Amita Kamath, Jack Hessel, and Kai-Wei Chang. What's "up" with vision-language models? investigating their struggle with spatial reasoning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 9161–9175, 2023.

Phillip Y Lee, Jihyeon Je, Chanho Park, Mikaela Angelina Uy, Leonidas Guibas, and Minhyuk Sung. Perspective-aware reasoning in vision-language models via mental imagery simulation. *arXiv preprint arXiv:2504.17207*, 2025.

Bo Li, Yuanhan Zhang, Dong Guo, Renrui Zhang, Feng Li, Hao Zhang, Kaichen Zhang, Yanwei Li, Ziwei Liu, and Chunyuan Li. Llava-onevision: Easy visual task transfer, 2024. URL `https://arxiv.org/abs/2408.03326`.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.

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

Wufei Ma, Haoyu Chen, Guofeng Zhang, Yu-Cheng Chou, Celso M de Melo, and Alan Yuille. 3dsrbench: A comprehensive 3d spatial reasoning benchmark. *arXiv preprint arXiv:2412.07825*, 2024.

Arjun Majumdar, Anurag Ajay, Xiaohan Zhang, Pranav Putta, Sriram Yenamandra, Mikael Henaff, Sneha Silwal, Paul Mcvay, Oleksandr Maksymets, Sergio Arnaud, Karmesh Yadav, Qiyang Li, Ben Newman, Mohit Sharma, Vincent Berges, Shiqi Zhang, Pulkit Agrawal, Yonatan Bisk, Dhruv Batra, Mrinal Kalakrishnan, Franziska Meier, Chris Paxton, Sasha Sax, and Aravind Rajeswaran. Openeqa: Embodied question answering in the era of foundation models. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

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

T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. doi: 10.1109/TSE.1976.233837.

Boyu Mi, Hanqing Wang, Tai Wang, Yilun Chen, and Jiangmiao Pang. Evolving symbolic 3d visual grounder with weakly supervised reflection. *arXiv preprint arXiv:2502.01401*, 2025.

Luigi Piccinelli, Yung-Hsu Yang, Christos Sakaridis, Mattia Segu, Siyuan Li, Luc Van Gool, and Fisher Yu. Unidepth: Universal monocular metric depth estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10106–10116, 2024.

Cheng Qian, Chi Han, Yi R Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. *arXiv preprint arXiv:2305.14318*, 2023.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, et al. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*, 2025.

Fan-Yun Sun, Shengguang Wu, Christian Jacobsen, Thomas Yim, Haoming Zou, Alex Zook, Shangru Li, Yu-Hsin Chou, Ethem Can, Xunlei Wu, et al. 3d-generalist: Self-improving vision-language-action models for crafting 3d worlds. *arXiv preprint arXiv:2507.06484*, 2025.

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

Peter Tong, Ellis Brown, Penghao Wu, Sanghyun Woo, Adithya Jairam Vedagiri IYER, Sai Charitha Akula, Shusheng Yang, Jihan Yang, Manoj Middepogu, Ziteng Wang, et al. Cambrian-1: A fully open, vision-centric exploration of multimodal llms. *Advances in Neural Information Processing Systems*, 37:87310–87356, 2024.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Yang Fan, Kai Dang, Mengfei Du, Xuancheng Ren, Rui Men, Dayiheng Liu, Chang Zhou, Jingren Zhou, and Junyang Lin. Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution. *arXiv preprint arXiv:2409.12191*, 2024a.

Zhiruo Wang, Daniel Fried, and Graham Neubig. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks. *arXiv preprint arXiv:2401.12869*, 2024b.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks. *arXiv preprint arXiv:2504.06821*, 2025.

Haoning Wu, Xiao Huang, Yaohui Chen, Ya Zhang, Yanfeng Wang, and Weidi Xie. Spatialscore: Towards unified evaluation for multimodal spatial understanding. *arXiv preprint arXiv:2505.17012*, 2025.

Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. C-pack: Packaged resources to advance general chinese embedding, 2023.

Jihan Yang, Shusheng Yang, Anjali W Gupta, Rilyn Han, Li Fei-Fei, and Saining Xie. Thinking in space: How multimodal large language models see, remember, and recall spaces. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 10632–10643, 2025.

Kaiyu Yang, Olga Russakovsky, and Jia Deng. Spatialsense: An adversarially crowdsourced benchmark for spatial relation recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 2051–2060, 2019.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R Fung, Hao Peng, and Heng Ji. Craft: Customizing llms by creating and retrieving from specialized toolsets. *arXiv preprint arXiv:2309.17428*, 2023.

Zhihao Yuan, Jinke Ren, Chun-Mei Feng, Hengshuang Zhao, Shuguang Cui, and Zhen Li. Visual programming for zero-shot open-vocabulary 3d visual grounding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 20623–20633, 2024.

Zheyuan Zhang, Fengyuan Hu, Jayjun Lee, Freda Shi, Parisa Kordjamshidi, Joyce Chai, and Ziqiao Ma. Do vision-language models represent space and how? evaluating spatial frame of reference under ambiguities. *arXiv preprint arXiv:2410.17385*, 2024.

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, et al. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

A The Use of Large Language Models (LLMs)

In this paper, we used LLMs only for grammar checking and light editing of the manuscript to polish the writing.

B Further Comparison with Related Tool-Use Work

Complementing §4.2, here we discuss in more depth how TVP's transductive approach fundamentally differs from representative methods in tool use and discovery.

TVP vs. Skillweaver. Skillweaver (Zheng et al., 2025) follows a **purely inductive** approach similar to VADAR: it speculatively proposes potentially useful functions before attempting any problem-solving experience, then synthesizes artificial test cases to validate these speculated functions. In contrast, TVP's tool creation is grounded in actual problem-solving experience, as TVP first accumulates experience solving problems, then parameterizes this experience into new tools, guaranteeing usefulness because each new tool encapsulates concrete, tested program solution patterns. Moreover, while Skillweaver **relies on a human-defined curriculum** with a predefined task order (procedural → navigational → information-seeking) tailored to the WebArena (Zhou et al., 2023) evaluation environment, TVP is **prior-free** and allows random ordering of datapoints, making it more generally applicable (see §D.1 for analysis of TVP's resilience to randomness).

TVP vs. ASI. While ASI (Wang et al., 2025) shares our key insight of creating tools from concrete program solutions, TVP differs in three critical aspects. First, ASI abstracts every episode individually, so each new tool simply represents one single episode. TVP abstracts a cluster of similar queries' solutions together, ensuring **each new tool generalizes over multiple example solutions** for better reusability. Second, ASI lacks library maintenance and cannot handle overlapping skills, leading to redundancy. TVP includes **explicit library maintenance** that merges similar skills, keeping the tool library clean and concise. Third, ASI extracts multiple tools from a single action trajectory, proposing multiple useful functions from one solution rather than abstracting the whole solution itself. TVP directly lifts an entire cluster of program solutions into a single abstract tool through transductive parameterization.

TVP and NePTune. NePTune (Kamali & Kordjamshidi, 2025) focuses on combining programmatic control flow with **symbolic logic operators** to enhance program expressiveness and execution. TVP's contribution is orthogonal, focusing on **novel tool creation** to enable more expressive programs through abstraction.

C Implementation Details

C.1 VADAR Reproducing Configurations

We directly utilized VADAR's official codebase and adhered to the official hyperparameter settings throughout, including: random batches of 15 questions for API proposal, GroundingDINO-SwinT-OGC (Liu et al., 2024) for object detection and UniDepth-v2-ViTS14 (Piccinelli et al., 2024) for depth estimation – the exact same tools used in our TVP implementation.

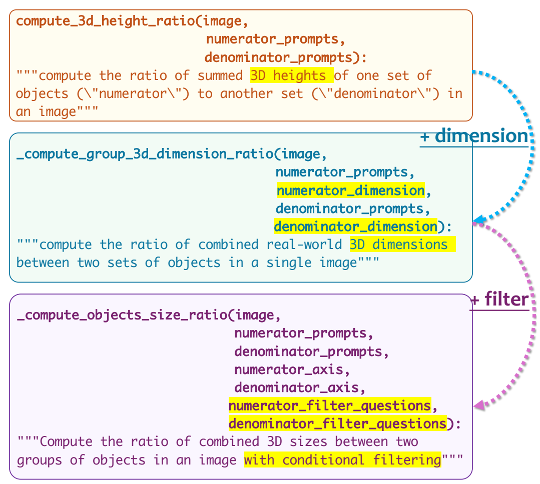

Figure 11: Hierarchical evolution of tool abstractions through transductive learning. From concrete step-by-step solutions (Level-0), TVP progressively abstracts more general and powerful tools through clustering (Level-1) and merging (Level-2), mirroring how human programmers build reusable function

C.2 TVP Configurations

We run the TVP pipeline on the Omni3D-Bench (§3.1) with the following configurations:

For the main pipeline (Alg. 1), we process all $N = 501$ questions from the entire dataset over $T = 3$ iterations. During each iteration, we generate $m = 4$ program candidates per question and retrieve $k = 3$ similar examples from the example library $\mathcal{E}$ using BGE-Large-En-v1.5 embeddings (Xiao et al., 2023) with a embedding similarity threshold $\tau_{\text{sim}} = 0.8$. Programs are accepted into the example library only if their quality score exceeds $\tau_q = 8.5$ on a 10-point scale. The tool abstraction process (Alg. 2) is triggered continuously after every $n_a = 1$ step (effectively at every step). We cluster examples using a similarity threshold of $\tau_{\text{sim}} = 0.8$ and require a minimum cluster size of $\tau_{\text{cluster}} = 4$ examples. Clusters with an abstraction potential score above $\tau_{\text{potential}} = 9.0$ are considered for tool creation. The validation process (Alg. 3) requires a minimum execution success rate of 100% and a correctness rate of at least 85% for divergent results. Both abstraction and program rewriting allow up to $R_{\max} = 2$ and $R_{\text{rewrite}} = 2$ retry attempts respectively. Tool deduplication (Alg. 4) is also performed after every $n_d = 1$ step. Tools are considered duplicates when their similarity exceeds 0.95. The merge process allows up to $R_{\text{merge}} = 2$ retry attempts to create a unified tool that passes validation.

Throughout our experiments, we maintain a **uniform random seed** of 42 across all pipeline components, governing aspects such as datapoint order (discussed more in §D.1).

In our main experiments (Tab. 1), we employ GPT-4o as the backbone program generator (LLM_{prog}), and as the VLM-based quality judge ($\text{LLM}_{\text{judge}}$) & correctness validator ($\text{LLM}_{\text{correct}}$). We use the reasoning model o4-mini for clustering ($\text{LLM}_{\text{cluster}}$), abstraction ($\text{LLM}_{\text{abstract}}$), deduplication ($\text{LLM}_{\text{dedup}}$), merging ($\text{LLM}_{\text{merge}}$), and program rewriting tasks. In our ablations on the scaling behavior (Fig. 7), we switch to Qwen2.5-Coder-Instruct-7/14/32B (Hui et al., 2024) for the backbone program generation, and run TVP for $T = 1$ iteration. Results discussed in §3.1 demonstrate TVP's **robustness to the backbone LLM**, as well as the **clear scaling trend with model capacity**.

Unless required by specific reasoning models like o4-mini (temperature = 1.0), LLM **temperatures** are set to 0.0 for deterministic tasks (quality judgment and correctness validation), ensuring rigorous assessment; and 1.0 for more creative tasks (program generation, abstraction, and rewriting), increasing the likelihood of finding better solutions.

In Fig. 4 (panel a), we use McCabe's Cyclomatic Complexity Number (CCN) (McCabe, 1976) as the program complexity measure, computed via the Lizard library [2], following the practice in Yuan et al. (2023).

C.3 Criteria in TVP's Judge Components

The program quality judge (§2.2) gates the admission to TVP's Example Library through evaluation across five comprehensive dimensions (as shown in Prompt 1): (1) 3D spatial understanding, following Marsili et al. (2025)'s official implementation for 3D concepts and definitions; (2) answer correctness with visual verification against the provided image; (3) appropriate program tool usage; (4) code quality including readability and efficiency; and (5) robustness to edge cases. These dimensions align with the critical requirements of both spatial reasoning and programming. The reliability of our quality judge is empirically validated in Tab. 1, where **enabling only the Example-Library in TVP already outperforms all baselines**. This demonstrates accurate admission of high-quality solutions in our Example Library that provide strong in-context examples.

The criteria for **tool abstraction potential** can be found in Prompt 2, which analyzes a group of program solutions clustered via question embeddings (embedding similarity is the first step of clustering, refer to §2.3). The abstraction potential focuses on general code abstraction requirements: (1) common computational patterns; (2) logical flow; (3) generalization capability; and (4) parameterization potential. We allow this flexibility in tool abstraction to **enable more diverse exploration of higher-level tools**, while still ensuring new tools' quality through the rigorous validation against all examples in the cluster before Tool Library admission (refer to §2.3).

C.4 Complexity Rating of 3D Spatial Reasoning Questions

In both our complexity-grouped evaluation illustrated in Figs. 5 and 6), and the curriculum-ordered TVP run discussed in §D.1, we use the **question complexity scores** rated via GPT-5 (high reasoning effort) with the prompt given in Prompt 3. The complexity rating evaluates questions along five axes, considering *e.g.*, 3D understanding; single-/multi-object relationships and multi-step reasoning; cognitive and computational load. Based on these scores on the scale of 1.0–10.0, we partition questions into three complexity buckets: Easy (1–3), Medium (4–6), and Hard (7–10).

D Additional Empirical Analyses and Discussion

D.1 TVP's Resilience to Random Datapoint Ordering

TVP is designed to operate on the fly without any dataset-specific priors, unlike previous methods such as Skillweaver (Zheng et al., 2025) that depends on human-defined curriculum for structured progression (see also discussion in §B). To validate our prior-free design choice, we compare the

[2] https://github.com/terryyin/lizard

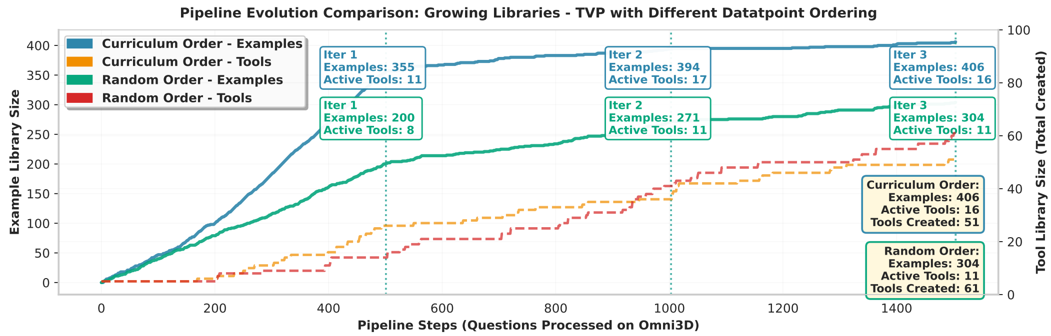

Figure 12: Pipeline evolution comparison between curriculum-ordered (as defined in §D.1) and random-ordered (as in our main experiments §3.1) datapoint processing. While curriculum ordering enables faster initial accumulation of examples and tools, random ordering ultimately creates more diverse tools through broader exploration of the problem space.

original TVP with **random ordering** (as used in our main experiments, §3.1) to **curriculum ordering** based on easy-to-hard progression through question complexity scores (details in §C.4).

Despite the intuition that starting experience with simpler problems, then gradually attempting harder problems seems a natural fit, we show in Fig. 13 that the overall performance is mostly on-par (both outperforming baselines), with the randomly-ordered TVP gradually getting better than the curriculum-ordered variant.

To understand this result, Fig. 12 reveals the evolution dynamics under both ordering strategies. The curriculum prior introduces two notable early-stage effects. First, it enables **earlier example accumulation**: datapoints with similar complexity clustered together facilitate more frequent example retrieval, resulting in 355 accumulated examples versus 200 with random ordering at the end of iteration 1. Second, it promotes **earlier tool creation**, as similar and simpler examples form eligible clusters sooner, yielding 11 active tools compared to 8 with random ordering after the first iteration.

However, **random ordering proves more beneficial for sustained library growth**. By encountering datapoints of varying complexities and patterns throughout processing, TVP explores a more diverse solution space. Although initial accumulation may be slower, this diversity enables continued progression as both libraries capture richer patterns. By completion, random ordering generates 61 total tools compared to 51 with curriculum ordering, demonstrating the value of diverse exploration over structured progression.

The above analysis speaks for TVP's design choice of resilience to random datapoint ordering. First, it enables **truly on-the-fly** operation without requiring any dataset-specific priors. Second, the diverse exploration inherent to random ordering fosters greater variety in accumulated experiences, leading to **more comprehensive tool creation** that better covers the problem space. This mirrors human learning that benefits from exposure to varied challenges rather than strictly structured curricula.

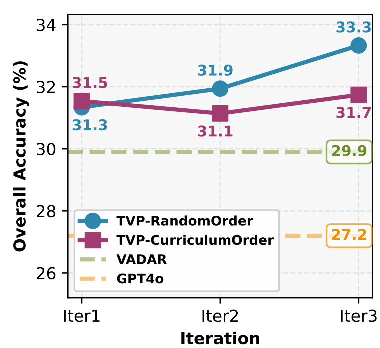

Figure 13: Performance comparison between TVP runs with random vs. curriculum-based datapoint ordering.

D.2 Computational Cost and Efficiency

We detail computational requirements for running TVP, as our transductive system should be highly accessible for research and deployment.

Cost structure and runtime. TVP operates in two distinct stages with different cost profiles:

(1) *Building TVP's dual libraries from scratch* involves processing the test set on the fly (Omni3D-Bench in §3.1), analogous to training a model. This stage requires approximately \$80 per iteration with our GPT-4o + GPT-4o-mini configuration, equivalent to about \$0.16 per question per iteration.

(2) *Applying TVP's built libraries to solve questions* (SpatialScore-*Hard* collection in §3.2) has minimal cost, usually equivalent to a single GPT-4o call per query.

GPUs are strictly optional in both stages. When used, the system requires under 4GB VRAM only to store the basic vision tools (GroundingDINO (Liu et al., 2024) and UniDepth (Piccinelli et al., 2024)), which can also run on CPUs.

The runtime for building TVP's dual libraries (analogous to training) stands at approximately 7 hours per iteration with our current implementation that executes programs sequentially.

Efficiency optimizations. We implement several strategies to improve TVP's cost-efficiency when building its dual libraries:

(1) *Early exit in tool validation*: Abstracted tools must achieve 100% execution success and 85% correctness on their validation cluster as per our current configurations (§C.2). For instance, for a cluster of 7 examples, validation exits early – thus **avoiding unnecessary computation** – when any one example fails execution (100% requirement not met); or when two fail the correctness check ($5/7 = 71\%$, drops below 85% pass rate)

(2) *Easy resumability*: We maintain comprehensive state checkpoints, supporting TVP's pause and resume at any point.

(3) *Embedding bank*: Since question embeddings remain unchanged, we keep a persistent storage of embedding vectors that enables simple lookup when retrieving (§2.1) or clustering examples (§2.3).

(4) *Parallel program generation and quality judge*: We generate program candidates in parallel and batch the quality judging for all valid candidates to reduce run-time.

E PROMPT TEMPLATES

Prompt 1: Quality Judge

```
You are an expert judge evaluating the quality of a program that solves a 3D spatial reasoning
      problem with tools (functions). Your task is to assess the program based on specific
     criteria and assign a quality rating from 1.0 to 10.0.

## TASK OVERVIEW
### Question
question

### Program to Evaluate
```python
program_code
```

Execution Results
- **Status:** exec_status: success/failure
- **Final Answer:** answer_text
- **Tools Used:** used_tools
- **Execution Error:** execution_error if present

Execution Namespace (All Variables)
execution_namespace_text

EVALUATION FRAMEWORK
Visual Evidence Verification
You are provided with an image of the scene. Use this visual evidence throughout your
     evaluation to:
- Verify the program's approach aligns with what's visible in the image
- Validate the answer's reasonableness based on visual proportions
- Check if 3D spatial relationships are correctly interpreted
- Confirm intermediate results match the visual scene
- Ensure all verification considers 3D content, not just 2D positioning

Namespace Analysis Considerations
When reviewing the execution namespace, specifically check for:
- Correct object identification (right object selected from multiple candidates)
- Proper bounding box matching
```

```
- Expected intermediate calculations
- Appropriate object filtering
- Correct depth-based 3D conversions
- Mismatched bounding boxes or coordinates
- Unexpected intermediate calculation results
- Objects that should have been filtered but weren't

### 3D Spatial Concepts & Definitions
**Core Definitions:**
- **Coordinate system:** (width, height, length) = (x, y, z) axis
- **Depth:** Distance from camera (smaller depth = closer to camera)
- **2D measurements:** Size/distance in pixel space (image coordinates)
- **3D measurements:** Size/distance in real world
- **3D size formula:** `3D size = 2D size * depth`
- **2D distance formula:** Euclidean distance between object center coordinates: `((x1-x2)**2
    + (y1-y2)**2)**0.5`
- **3D distance formula:** `3D_distance = (2D_distance**2 + (depth1 - depth2)**2)**0.5`
- **Distance to camera:** Simply the object's depth value

**Key Considerations:**
- 2D sizes are in pixel space. To convert to 3D size, multiply by depth
- Objects with same 2D dimensions but different depths have different 3D sizes
- 3D distance requires the Pythagorean formula combining 2D distance and depth difference - as
     defined above
- Center coordinates should determine "leftmost"/"rightmost"
- The `loc()` function should not handle compound descriptions - must locate base objects then
     filter for the desired condition
- All objects satisfying a condition must be checked, not just the first
- Multiple objects with same property values require proper tie-breaking
- Hypothetical object counts (e.g., "if a table has X legs") require counting actual objects
    in image

## RATING CRITERIA (1.0 - 10.0 Scale)
### 1. **3D Spatial Understanding**
- Properly converts between 2D and 3D measurements
- Correctly handles 3D size/distance calculation
- Correctly uses center coordinates for distance calculations and leftmost/rightmost
    determinations
- Interprets spatial relationships correctly (e.g., "largest" means 3D, not 2D)
- Answer is visually verifiable and reasonable

### 2. **Correctness and Visual Verification**
- Solves the problem correctly based on the actual image
- Aligns with visual evidence from the image
- Intermediate results are consistent with visible scene
- Spatial relationships match visual reality

### 3. **Tool Usage Efficiency**
- Uses appropriate tools for the task
- Leverages higher-level "learned" tools when suitable
- Avoids reimplementing existing functionality
- Note: Basic tools are acceptable when no higher-level tools fit

### 4. **Code Quality**
- Well-structured with clear variable names
- Follows tool usage patterns correctly
- Efficient without unnecessary operations
- Includes helpful comments

### 5. **Robustness and Edge Cases**
- Properly filters located objects for properties rather than using complex `loc()` queries
- Handles multiple objects with same property (proper tie-breaking)
- Manages empty lists and None values appropriately
- Manages container relationships (e.g., "in", "on") properly
- Includes appropriate error checking

## REQUIRED OUTPUT FORMAT
You MUST provide your response in exactly this format:

<rating>NUMBER</rating>
<reasoning>
[Detailed explanation covering:
- How visual evidence supports/contradicts the program's logic
- Specific strengths and weaknesses identified
- Analysis of 3D spatial reasoning approach
- Evaluation of intermediate execution results
- Missed opportunities to use available tools
- Overall assessment based on all criteria]
</reasoning>
Where NUMBER is a decimal between 1.0 and 10.0.
```

```
---

## APPENDIX: Available Tools Reference

The program had access to tool_counts tools total: num_basic_tools basic tools and
     num_created_tools learned tools.

### Basic Tools (Level 0)
tool_signature, tool_docstring

### Learned Tools (Level 1+)
tool_signature, tool_docstring
```

Prompt 2: Abstraction Potential Analysis

````
You are an expert at analyzing visual reasoning programs to identify common patterns that
     could be abstracted into reusable functions.

## Your Task
Analyze num_cluster_examples visual reasoning examples to:
1. Identify common computational patterns across examples
2. Group them into clusters based on shared functionality
3. Rate each cluster's abstraction potential (0-10 scale)

## Examples to Analyze
examples (question, program solution)

## Clustering Criteria
Identify clusters based on:
1. **Common computational patterns** - e.g., finding largest/smallest, counting with
     conditions
2. **Similar operations sequence** - e.g., locate -> filter -> compute -> compare
3. **Shared logic structure** - e.g., iteration patterns, comparison logic
4. **Abstractable functionality** - can be parameterized into a reusable function

## Evaluation Requirements
### For Each Cluster Provide:
- **Example IDs** that belong to it
- **Common pattern** explanation
- **Parameters** that vary between examples
- **Abstraction potential rating** (0-10) based on:
  * How well the pattern generalizes
  * Parameter variability coverage
  * Clarity of the abstraction
  * Reusability across similar tasks
- **Reasoning** for the rating

### Critical Constraints
- **Each example must belong to exactly ONE cluster or be marked as unclustered**
- Focus on computational patterns, not surface similarities
- Only create clusters with strong shared patterns

## Response Format
Provide your analysis using this exact format. Include as many cluster blocks as needed,
     followed by an optional unclustered block:

```
<cluster>
<example_ids>[comma-separated list of example IDs]</example_ids>
<pattern>[Description of the common computational pattern]</pattern>
<parameters>[List of parameters that vary between examples]</parameters>
<abstraction_potential>[Rating from 0-10]</abstraction_potential>
<reasoning>[Explanation for the rating and how the pattern could be abstracted]</reasoning>
</cluster>

[Additional <cluster> blocks as needed...]

<unclustered>
<example_ids>[comma-separated list of example IDs that don't fit clusters]</example_ids>
<reasoning>[Explanation of why these examples don't cluster well]</reasoning>
</unclustered>
```

**Remember:** Every example ID must appear in exactly ONE cluster or in the unclustered group.
````

Prompt 3: Question Complexity Rating

```
You are an expert in evaluating the complexity of 3D spatial reasoning questions. Your task is
     to assign a complexity score (1.0 - 10.0 scale) to a single question based on its
    inherent spatial reasoning difficulty.

## QUESTION TO EVALUATE

**Question:** question

**Answer Type:** answer type: float/integer/multiple-choice/etc.

## EVALUATION FRAMEWORK

### 1. **3D Spatial Reasoning Requirements**
- Does the question require understanding of 2D (pixel/image space) vs 3D (real-world)
    measurements?
- Does it involve depth understanding and distance-from-camera concepts?
- Does it require 3D size calculations or understanding that same 2D size at different depths
    means different 3D sizes?
- Does it involve 3D distance calculations (combining 2D distance and depth differences)?
- Does it require converting between measurement spaces?

### 2. **Spatial Relationship Complexity**
- How many objects are involved in the spatial reasoning?
- Types of relationships:
  - Simple property identification (color, material, count)
  - Spatial relationships (distance, size comparison, relative position)
  - Complex spatial relationships (e.g., "to the right of X and behind Y")
- Does it require multi-step reasoning with intermediate conclusions?
- Comparative judgments vs. absolute measurements

### 3. **Cognitive Load and Constraints**
- Number of constraints or conditions that must be simultaneously satisfied
- Need to identify and distinguish between multiple candidate objects
- Hypothetical or conditional reasoning ("if X is Y meters, then...")
- Handling of multiple objects with potentially ambiguous descriptions
- Container relationships (objects "in" or "on" other objects)

### 4. **Calculation and Quantitative Complexity**
- Simple identification or counting vs. numerical calculations
- Ratio, proportion, or percentage calculations
- Multiple measurement comparisons
- Distance or size computations requiring formulas
- Precision requirements

### 5. **Answer Type Indicators**
- **yes/no (binary):** Often simpler verification tasks but can be complex depending on what's
     being verified
- **multiple choice (str with options):** Requires discrimination among bounded options
- **numerical (float/int):** Often requires precise calculations and measurements
- **open string:** May require identification and categorization

## COMPLEXITY SCORING GUIDELINES

Consider the full spectrum of complexity:

**Lower end:** Simple, direct questions requiring minimal spatial reasoning
- Single object property identification
- Basic counting
- Simple yes/no verification with clear criteria

**Middle range:** Moderate spatial reasoning and calculation
- Size or distance comparisons between pairs of objects
- Simple ratio calculations
- Object identification with multiple constraints
- Basic 2D-to-3D conversions

**Higher end:** Complex multi-step spatial reasoning
- Multiple object comparisons with numerous constraints
- Complex calculations involving combined measurements
- Hypothetical reasoning with conditional calculations
- Spatial relationships involving many objects with interdependencies
- Ratios of combined or derived quantities

Assign a score on the scale of 1.0 - 10.0 based on the question's position in this complexity
    spectrum. Consider ALL evaluation dimensions together.

## REQUIRED OUTPUT FORMAT

Provide your response in EXACTLY this format:
```

```
<score>X.X</score>
<reasoning>
[Detailed explanation covering:
- Which evaluation dimensions contribute most to complexity
- Specific aspects that increase or decrease difficulty
- Why this score is appropriate
- Key spatial reasoning challenges in the question]
</reasoning>

The score should be a decimal number between 1.0 and 10.0. Use your judgment to place the
     question appropriately on the complexity spectrum.
```

F Complete TVP Algorithm

Algorithm 1 Transductive Visual Programming (TVP) Pipeline

Input: Dataset $\mathcal{D} = \{(I_i, q_i)\}_{i=1}^{N}$ (images, questions)

Initialize: Example Library $\mathcal{E} \leftarrow \emptyset$, Tool Library $\mathcal{T} \leftarrow$ {predefined tools}
Parameters: quality threshold τ_q, abstraction interval n_a, deduplication interval n_d
for iteration $t = 1$ to T **do**
  for each question $q_i \in \mathcal{D}$ **do**
    # Step 1: Retrieve similar examples
    $\mathcal{E}_{\text{sim}} \leftarrow \text{RetrieveSimilar}(\mathcal{E}, q_i, k = 3)$ ▷ Exclude q_i itself
    # Step 2: Generate program candidates
    $\mathcal{C} \leftarrow \emptyset$
    for $j = 1$ to m **do** ▷ m candidates per question
      $p_j \leftarrow \text{LLM}_{\text{prog}}(q_i, \mathcal{E}_{\text{sim}}, \mathcal{T})$ ▷ In-context learning
      $\mathcal{C} \leftarrow \mathcal{C} \cup \{p_j\}$
    end for
    # Step 3: Execute and filter
    $\mathcal{C}_{\text{succ}} \leftarrow \emptyset$
    for each $p_j \in \mathcal{C}$ **do**
      $\text{result}_j, \text{namespace}_j \leftarrow \text{Execute}(p_j, I_i, \mathcal{T})$
      if $\text{result}_j \neq \text{None} \wedge \neg\text{error}$ **then**
        $\mathcal{C}_{\text{succ}} \leftarrow \mathcal{C}_{\text{succ}} \cup \{(p_j, \text{result}_j, \text{namespace}_j)\}$
      end if
    end for
    # Step 4: Judge quality with vision model
    for each $(p_j, \text{result}_j, \text{namespace}_j) \in \mathcal{C}_{\text{succ}}$ **do**
      $\text{quality}_j \leftarrow \text{LLM}_{\text{judge}}(q_i, p_j, \text{namespace}_j, I_i)$ ▷ 1-10 scale
    end for
    # Step 5: Select best and update library
    $p^*, \text{quality}^*, \text{namespace}^* \leftarrow \arg\max_j \text{quality}_j$
    if $\text{quality}^* \geq \tau_q$ **then**
      $e \leftarrow \text{Example}(q_i, p^*, \text{quality}^*, \text{namespace}^*)$
      if $\exists e' \in \mathcal{E}$ with $e'.q = q_i$ **then** ▷ Update existing
        if $\text{quality}^* > e'$.quality or different tools used **then**
          $\mathcal{E} \leftarrow (\mathcal{E} \setminus \{e'\}) \cup \{e\}$
        end if
      else
        $\mathcal{E} \leftarrow \mathcal{E} \cup \{e\}$ ▷ Add new
      end if
    end if
    # Step 6: Abstraction interval
    if $|\mathcal{E}| \bmod n_a = 0$ **then**
      $\mathcal{T} \leftarrow \text{AbstractTools}(\mathcal{E}, \mathcal{T})$ ▷ See Algorithm 2
    end if
    # Step 7: Deduplication interval
    if $|\mathcal{E}| \bmod n_d = 0$ **then**
      $\mathcal{T} \leftarrow \text{DeduplicateTools}(\mathcal{T})$ ▷ See Algorithm 4
    end if
  end for
end for
return $\mathcal{E}, \mathcal{T}$

Algorithm 2 AbstractTools - Tool Abstraction from Example Clusters

```
Input: Example Library $\mathcal{E}$, Tool Library $\mathcal{T}$
Parameters: similarity threshold $\tau_{\text{sim}} = 0.8$, cluster size threshold $\tau_{\text{cluster}} = 4$, potential threshold $\tau_{\text{potential}} = 9.0$
# Step 1: Filter eligible examples
$\mathcal{E}_{\text{eligible}} \leftarrow \{e \in \mathcal{E} : e.\text{status} \neq \text{"abstracted"}\}$
# Step 2: Cluster by similarity
$\mathcal{G} \leftarrow \text{ClusterBySimilarity}(\mathcal{E}_{\text{eligible}}, \tau_{\text{sim}})$
for each cluster $G \in \mathcal{G}$ with $|G| \geq \tau_{\text{cluster}}$ do
    # Step 3: Analyze cluster for patterns
    pattern, potential $\leftarrow \text{LLM}_{\text{cluster}}(G)$
    if potential $\geq \tau_{\text{potential}}$ then                ▷ Abstraction potential threshold
        # Step 4: Create tool with retry
        for retry $= 1$ to $R_{\max}$ do
            if retry $= 1$ then
                $t \leftarrow \text{LLM}_{\text{abstract}}(G, \text{pattern}, \mathcal{T})$
            else
                $t \leftarrow \text{LLM}_{\text{abstract}}(G, \text{pattern}, \mathcal{T}, \text{feedback})$
            end if
            # Step 5: Validate tool
            val $\leftarrow \text{ValidateTool}(t, G, \mathcal{T})$                ▷ See Algorithm 3
            if val.passed then
                $\mathcal{T} \leftarrow \mathcal{T} \cup \{t\}$
                # Update examples with new tool
                for each $e \in G$ with successful rewrite do
                    $e$.program $\leftarrow$ val.rewritten[$e$]
                    $e$.status $\leftarrow$ "abstracted"
                    $e$.tools_used $\leftarrow e$.tools_used $\cup \{t\}$
                end for
                break
            else
                feedback $\leftarrow$ val.errors                ▷ For retry
            end if
        end for
    end if
end for
return $\mathcal{T}$
```

Algorithm 3 ValidateTool - Two-Stage Tool Validation

Input: Tool t, Examples G, Tool Library $\mathcal{T}$
Output: Validation result with rewritten programs

Stage 1: Execution validation
successes $\leftarrow 0$, rewrites $\leftarrow \{\}$
for each example $e \in G$ **do**
    for retry $= 1$ to R_{rewrite} **do**
        $p' \leftarrow \text{RewriteProgram}(e.\text{program}, t)$
        $\text{result}', \text{namespace}' \leftarrow \text{Execute}(p', e.\text{image}, \mathcal{T} \cup \{t\})$
        if $\text{result}' \neq \text{None} \wedge \neg\text{error}$ **then**
            successes $\leftarrow$ successes $+ 1$
            $\text{rewrites}[e] \leftarrow (p', \text{result}', \text{namespace}')$
            break
        end if
    end for
    if $\text{successes}/|G| < 1.0$ **then** ▷ Early exit
        return $\{\text{passed} : \text{False}, \text{errors} : \text{execution\_failures}\}$
    end if
end for
# Stage 2: Correctness validation for divergent results
correct $\leftarrow 0$, divergent $\leftarrow 0$
for each $e \in G$ with successful rewrite **do**
    if $\text{rewrites}[e].\text{result} \neq e.\text{result}$ **then**
        divergent $\leftarrow$ divergent $+ 1$
        $\text{verdict} \leftarrow \text{LLM}_{\text{correct}}(e, \text{rewrites}[e], e.\text{image})$
        if verdict = "CORRECT" **then**
            correct $\leftarrow$ correct $+ 1$
        end if
    end if
end for
$\text{overall\_correct} \leftarrow (|G| - \text{divergent} + \text{correct})/|G|$
if $\text{overall\_correct} \geq 0.85$ **then**
    return $\{\text{passed} : \text{True}, \text{rewrites} : \text{rewrites}\}$
else
    return $\{\text{passed} : \text{False}, \text{errors} : \text{correctness\_failures}\}$
end if

Algorithm 4 DeduplicateTools - Merge Similar Tools

Input: Tool Library $\mathcal{T}$

```
1: # Filter eligible tools
2: T_eligible ← {t ∈ T : t.level > 0 ∧ ¬t.deprecated}
3: # Find duplicate groups
4: M ← LLM_dedup(T_eligible)                          ▷ Groups with similarity ≥ 0.95
5: for each merge group M ∈ M do
6:     # Get examples using these tools
7:     E_M ← {e ∈ E : ∃t ∈ M, t ∈ e.tools_used}
8:     for retry = 1 to R_merge do
9:         if retry = 1 then
10:            t_merged ← LLM_merge(M, strategy)
11:        else
12:            t_merged ← LLM_merge(M, strategy, feedback)
13:        end if
14:        val ← ValidateTool(t_merged, E_M, T)
15:        if val.passed then
16:            T ← T ∪ {t_merged}
17:            for each t ∈ M do
18:                t.deprecated ← True
19:                t.reason ← "Merged into t_merged"
20:            end for
21:            # Update examples
22:            for each e ∈ E_M with successful rewrite do
23:                Update e with merged tool
24:            end for
25:            break
26:        else
27:            feedback ← val.errors
28:        end if
29:    end for
30: end for
31: return T
```