PyVision: Agentic Vision with Dynamic Tooling

Shitian Zhao[1,§,*] **Haoquan Zhang**[1,3,*] **Shaoheng Lin**[1,*] **Ming Li**[1,*] **Qilong Wu**[4,*]
Kaipeng Zhang[1,5,†] **Chen Wei**[2,†]
[1]Shanghai AI Lab [2]Rice University [3]CUHK [4]NUS [5]SII

Project Page Inference Code PyVision Demo

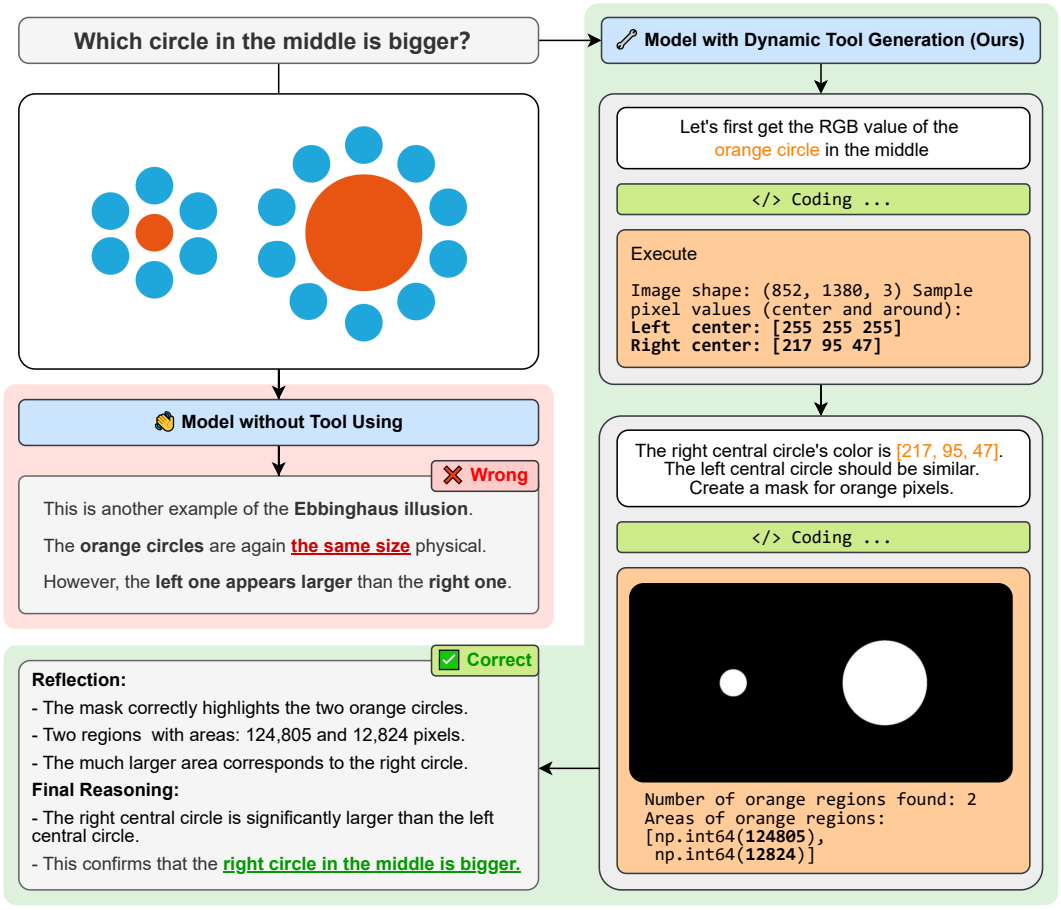

Figure 1: **Overcoming Illusory Heuristics with Code.** This visual puzzle mimics the well-known Ebbinghaus illusion [19], but with a twist: it reverses the typical size context, making the correct answer visually obvious to humans. Yet, a standard MLLM [35] mistakenly recalls the well-documented illusion template to answer "same size". In contrast, PyVision behaves agentically, probing pixel values, segmenting objects, and computing the actual sizes via on-the-fly Python code to reach the correct answer. This example highlights how dynamic tooling enables adaptive, grounded, verifiable visual reasoning beyond superficial pattern matching.

[*]Joint First Author; [§]Project Lead; [†]Corresponding Author

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Multi-Turn Interactions in Large Language Models.

Abstract

LLMs are increasingly deployed as agents, systems capable of planning, reasoning, and dynamically calling external tools. However, in visual reasoning, prior approaches largely remain limited by predefined workflows and static toolsets. In this report, we present `PyVision`, an interactive, multi-turn framework that enables MLLMs to autonomously generate, execute, and refine Python-based tools tailored to the task at hand, unlocking flexible and interpretable problem-solving. We develop a taxonomy of the tools created by `PyVision` and analyze their usage across a diverse set of benchmarks. Quantitatively, `PyVision` achieves consistent performance gains, boosting GPT-4.1 by +7.8% on V* and Claude-4.0-Sonnet by +31.1% on VLMsAreBlind-mini. These results point to a broader shift: dynamic tooling allows models not just to use tools, but to invent them, advancing toward more agentic visual reasoning.

1 Introduction

The idea of AI agents, systems that can autonomously plan and execute tasks, is rapidly gaining traction in modern AI research. Large language models (LLMs), originally built for text generation, have quickly evolved into capable agents that can formulate plans, interact with environments, and call external tools or functions to solve complex problems with minimal human oversight [34, 33, 32, 28, 40, 30, 29, 16, 14, 4, 27, 1]. But beyond simply using tools, the more profound leap lies in an agent's ability to invent them, such as dynamically generating code snippets tailored to its task or environment. This capacity to create problem-solving tools on the fly is not just powerful, but foundational to intelligence. As Benjamin Franklin remarked, "Man is a tool-making animal".

Interestingly, the idea of using external computational modules for complex reasoning is not new, particularly in the vision domain. Early works such as Neural Module Networks [2] introduced a parser that orchestrated a set of predefined functions, embracing a neuro-symbolic approach to visual reasoning. This line of work inspired a series of influential successors (Tab. 1). Unlike end-to-end models, these systems explicitly represent each reasoning step and producing transparent and inspectable intermediate outputs, offering a promising path for tackling complex and compositional visual reasoning.

However, prior works typically rely on predefined workflows and static toolsets within single-turn frameworks, limiting the flexibility, creativity, and adaptability that modern LLM agents can achieve through dynamic tooling. With the growing coding and reasoning capabilities of today's MLLMs, we can now move beyond these constraints in visual reasoning: models can dynamically generate code snippets in a multi-turn setup, building tools on the fly that are tailored to the task at hand.

Recent developments like OpenAI's "Thinking with Images" [37] highlight this potential, but they offer limited visibility into how this process actually works. In this report, we present and analyze how advanced MLLMs with strong coding abilities, in our case, GPT-4.1 [35] and Claude-4.0-Sonnet [3], can dynamically create and leverage Python-based visual tools. We introduce `PyVision`, an interactive framework in which the model autonomously generates, executes, and iteratively refines Python code in response to multimodal user queries. To support this dynamic tooling loop, we build on Python's rich ecosystem of mature libraries and carefully engineer both the system prompts and the runtime environment to enable seamless, multi-turn interaction between the MLLM and Python interpreter.

We then analyze the tools generated by `PyVision` in depth. To do so, we construct a taxonomy that classifies the tools into four broad categories: basic image processing, advanced image processing, visual prompting and sketching, and numerical and statistical analysis, alongside a long tail of creative, task-specific operations (Fig. 1). This framework enables us to examine how different benchmarks and domains elicit distinct patterns of tool usage. For instance, perception-heavy tasks often trigger operations like cropping and contrast enhancement, while math and logic benchmarks rely more on visual sketching and numerical analysis. These findings highlight the power of dynamic tool

Methods	Dynamic Workflow	Dynamic Tool Generation	Multi-Turn Framework
NMN [2]	×	×	×
IEP [20]	×	×	×
VisProg [12]	×	×	×
Visual ChatGPT [51]	×	×	✓
ViperGPT [47]	×	×	×
MM-REACT [56]	×	×	×
HuggingGPT [42]	×	×	×
Image-of-Thought [61]	×	×	×
Visual Sketchpad [18]	✓	×	✓
VAT [24]	×	×	×
`PyVision`	✓	✓	✓

Table 1: **Comparison** between `PyVision` and previous tool-using methods for visual reasoning.

generation: it equips the model with the flexibility to adapt its strategy to the unique demands of each task and domain.

Results across major benchmarks reveal that `PyVision` consistently improves the performance of strong backend models. Notable improvements include a +7.8% boost on V* [52] with `PyVision`-GPT-4.1, an +8.3% gain on Visual Puzzles [43], and a dramatic leap on VLMsAreBlind-mini [41], where `PyVision`-Claude-4.0-Sonnet improves from 48.1% to 79.2%, marking a remarkable +31.1% increase. Our results suggest that `PyVision` acts as an amplifier of the backend model's innate strengths: gaining more at perception tasks when paired with perceptually strong models like GPT-4.1, and at abstract reasoning when paired with Claude-4.0-Sonnet. In short, dynamic tooling does not override model capabilities. It unlocks them.

Ultimately, the agentic `PyVision` with dynamic tooling not only provides practical performance benefits, it also signals a broader shift in multimodal reasoning. By empowering models to invent new computational tools on the fly, we move closer to versatile, autonomous, and genuinely creative AI systems capable of adapting in real-world visual reasoning scenarios.

2 Related Work

Multi-Modal Tool Using. To solve the compositional Visual Question Answering (VQA) task in a more transparent and interpretable fashion, early work NMN [2] use a heuristic method while IEP [20] train an LSTM network as the program generator. In the era of LLMs, a pretrained LLM, *e.g.,* GPT-4, is used to generate programs.

Visual ChatGPT [51], MM-REACT [56], HuggingGPT [42], Image-of-Thought [61], and VAT [24] design workflows to process VQA inputs and produce final answers. In VisProg [12] and ViperGPT [47], researchers predefine a static toolset for specific vision tasks and prompt the LLMs or MLLMs to generate programs that invoke these tools to support reasoning. As LLMs' coding abilities improve, Visual Sketchpad [18] predefines a toolset and prompts the LLM to program and execute code on the fly, offering more flexibility. These prior works rely on a static toolset containing various visual parsers [10], *e.g.,* detection models (GroundingDINO [25]) and segmentation models (SAM [21]), which limits generality across vision tasks and makes the external models a bottleneck. In contrast, `PyVision` uses Python as the sole primitive tool. With the advanced coding and multi-modal understanding abilities of today's MLLMs, *e.g.,* Claude-4.0 [3] and GPT-4.1 [35], they can write Python code to construct and execute complex tools on the fly, enabling more general and flexible reasoning.

Thinking with Images. In o3's [37] blog, thinking with images is presented as an attractive feature. CoGCoM [39] synthesizes program-integrated data and teaches the MLLM to use predefined tools during inference. DeepEyes [60], Pixel Reasoner [44], OpenThinkIMG [46, 45], and Chain-of-Focus [58] incentivize MLLMs to develop the ability to "think with images using predefined tools" through reinforcement learning. In `PyVision`, we support thinking with images by using Python as the tool creation interface, enabling the MLLM to self-generate more complex and adaptive tools based on varying scenarios.

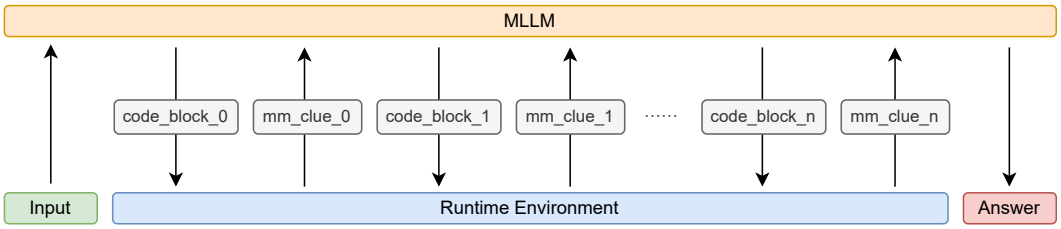

Figure 2: PyVision, an interactive and multi-turn framework capable of dynamic tool generation, designed for multimodal reasoning. In an inference session, PyVision performs n+1 interaction turns with the Python interpreter. In the figure, code_block_i refers to the generated Python code by the MLLM in the i-th turn, and mm_clue_i the executed multi-modal outputs by the Python interpreter. This loop continues until the MLLM outputs a final answer.

3 PyVision

We propose PyVision, an interactive, multi-turn framework for multimodal reasoning. PyVision empowers an MLLM with the ability to dynamically generate and execute Python code during inference. In each session, the MLLM receives an input, generates Python code in response, and executes it within an isolated Python runtime. The resulting output—textual, visual, or both—is fed back into the MLLM's context, allowing it to iterate and refine its reasoning over multiple turns until it produces a final answer.

Unlike prior approaches that rely on a fixed toolset, such as detection [25] or segmentation [21] models, PyVision provides only Python as building blocks for tools. This design leverages Python's rich ecosystem of scientific and vision libraries, for example, OpenCV [7], Pillow [8], NumPy [13], Pandas [31], Scikit-learn [38], and Scikit-image [48]. With access to such a versatile ecosystem, the model can generate highly adaptive tools tailored to diverse tasks.

System Prompt Design. To guide the MLLM's reasoning and code generation, PyVision uses a carefully constructed system prompt in addition to user queries. The system prompts encode operational instructions that specify how to access input images, structure code, and return final answers. Key components include:

- Encouraging the MLLM to generate code to solve the task.
- Input images or video frames are pre-loaded as variables named image_clue_i, where i denotes the image index. This allows the model to reference the images without additional loading code. We also provide image resolution that helps operations like cropping.
- Output from the code is expected via specific functions: print() for textual results and plt.show() for image visualizations.
- Each generated code block is wrapped in a <code> tag to enable reliable parsing.
- Final answers are enclosed in a <answer> tag for consistent evaluation.

With this design, the two MLLMs we experiment with, GPT-4.1 [35] and Claude-4.0-Sonnet [3], can reliably generate parsable and executable code blocks that rarely crash. The full system prompt is included in appendix A.

Multi-Turn Interaction between Runtime and the MLLM. As illustrated in Fig. 2, PyVision operates as a multi-turn agentic loop between the MLLM and an isolated Python runtime. In the i-th turn, the MLLM generates a code block code_block_i, which is executed to produce multimodal results mm_clue_i. These results are appended to the MLLM's context, enabling it to update its reasoning in the next turn. This loop continues until the MLLM automatically decides to output a final boxed answer.

To support robust and effective multi-turn interaction between the MLLM and the runtime environment of Python, PyVision incorporates several design principles:

- **Process isolation**: Each code snippet is executed in a subprocess dynamically spawned by the main process, ensuring that crashes or side effects in one execution do not impact the overall inference session.

- **Cross-turn persistence**: The runtime environment retains variables and state across turns. This allows the model to reuse or modify intermediate Python code execution results in previous turns, *e.g.,* first cropping an image, then applying filters, and finally computing geometric features to complete a task.
- **File-system safe I/O**: Communication between the runtime and the MLLM is handled through structured variable passing [9, 53, 11], guided by system prompts. This avoids direct dependencies on the host file system.

Together, these mechanisms enable `PyVision` to serve as a flexible, secure, and powerful platform for dynamic tool generation in multi-modal reasoning tasks.

4 Results on Versatile Benchmarks

Baselines. To evaluate `PyVision`'s effectiveness on diverse multi-modal scenarios, we test it on versatile benchmarks with MLLMs including GPT-4.1 [35] and Claude-4.0-Sonnet [3] as the backend. We use plain chain-of-thought prompting [50, 22] as our baseline. The inference parameter settings and the prompt details are in appendix A.

	MathVista	MathVision-mini	MMMU	VisualPuzzles	VLMsAreBlind-mini	V*
GPT-4o	61.4	–	68.7	41.1	–	73.9
o1	71.8	–	77.6	51.8	–	69.7
o3	86.8	–	82.9	54.0	–	95.7
GPT-4.1	69.9*	46.4	71.9*	44.9	67.1	68.1
`PyVision`-GPT-4.1	71.7 +1.8	48.7 +2.3	74.3 +2.4	47.4 +2.5	69.7 +2.6	75.9 **+7.8**
Claude-4.0-Sonnet	71.4	48.0	74.4	42.7	48.1	56.5
`PyVision`-Claude	76.2 +4.8	51.3 +3.3	74.6 +0.2	51.0 **+8.3**	79.2 **+31.1**	56.8 +0.3

Table 2: **Performance on six benchmarks**. Improvements over each base model appear beneath the scores. We highlight a **+7.8**% gain on V* by `PyVision`-GPT-4.1, **+8.3**% on VisualPuzzles and **+31.1**% on VLMsAreBlind-mini by `PyVision`-Claude. *GPT-4.1 results are self-collected with plain chain-of-though prompting (appendix A.2) in June 2025.

Results. Tab. 2 highlights how adding `PyVision`'s dynamic tooling consistently boosts two strong back-end models across a diverse benchmark suite. For GPT-4.1, `PyVision` yields uniform gains on every dataset, from modest improvements on math-centric tasks: +1.8% on MathVista and +2.4% on MMMU, to a sizeable +7.8% on the fine-grained visual-search benchmark V*. Claude-4.0-Sonnet shows a sharper pattern: while math and general-reasoning tasks improve by roughly +3% to +5%, symbolic-vision performance on VLMsAreBlind-mini soars by +31.1%. In short, dynamic tool generation delivers broad, task-dependent gains, which also depends on the backend model's capability, discussed next.

`PyVision` **Amplifies What the Backend MLLM Does Best, Reasoning or Perception.** To better understand the relationship between `PyVision`'s performance gains and the inherent strengths of backend models, we focus on two representative benchmarks: *MathVision-mini* [49], which emphasizes abstract reasoning, and *V** [52], which highlights perception ability. Claude-4.0-Sonnet, stronger in abstract reasoning as shown by its higher MathVision-mini performance (48.0% vs. 46.4% for GPT-4.1), experiences a larger boost from PyVision (+3.3%) compared to GPT-4.1's more modest gain (+2.3%). Conversely, GPT-4.1, superior in perceptual tasks like V* (68.1% vs. Claude-4.0-Sonnet's 56.5%), achieves a significantly greater improvement with `PyVision` (+7.8% vs. only +0.3%). This complementary pattern suggests that the effectiveness of dynamic tooling provided by `PyVision` depends critically on the backend model's foundational reasoning and perception strengths.

Further supporting this hypothesis, experiments with Qwen2.5-VL-72B [6] yield similar findings: weaker abstract reasoning capabilities (18.4% on MathVision-mini) lead to limited improvement (+1.7%), while stronger perceptual performance (67.0% on V*) translates into substantial gains (+10.0%). These insights underline that `PyVision` amplifies existing backend model strengths,

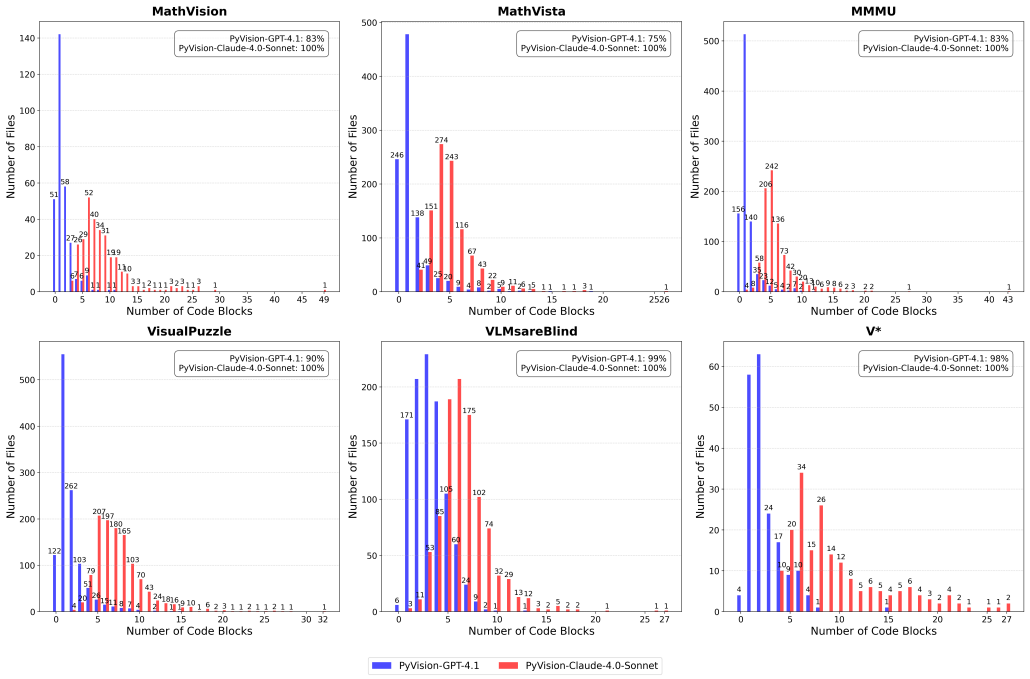

Figure 3: **Multi-Turn Interaction Patterns Across Tasks and Backend Models**. The histograms show the distribution of the number of generated code blocks per query across six benchmarks. `PyVision`-GPT-4.1 (blue) and `PyVision`-Claude-4.0-Sonnet (red) exhibit distinct interaction patterns, with Claude consistently generating code more frequently and with more turns. The legend in each subplot indicates the percentage of samples that involved at least one code block.

making the interplay of reasoning and perception crucial for unlocking the full potential of dynamic multimodal tooling.

How Often and How Much MLLMs Generate Code? Fig. 3 shows the distribution of the number of code blocks generated per user query across six benchmarks, comparing `PyVision` backed by GPT-4.1 and Claude-4.0-Sonnet. Each subplot visualizes how frequently the model uses code during multi-turn inference, with the legend indicating the percentage of query sessions that include any code generation. We observe that Claude-4.0-Sonnet consistently generates more code than GPT-4.1 across all domains, often with longer toolchains per query and reaching 100% code coverage. Conversely, GPT-4.1 tends to use fewer code blocks. These trends suggest a difference in agentic behavior, reflecting underlying differences in how each MLLM parses complexity and utilizes code to support reasoning.

5 Dynamically Generated Tools

Examples in Different Tasks and Domains. We start our analysis by presenting examples of `PyVision` across diverse tasks and domains in Figs. 6 to 10. These examples illustrate how `PyVision` autonomously creates task-specific and domain-specific tools tailored to each unique challenge, emerging voluntarily from `PyVision`'s multi-turn code generation and execution.

5.1 Tooling Taxonomy

To better understand the types of tools generated by `PyVision`, we construct a taxonomy based on the code it produces across various tasks and domains (Sec. 4). Specifically, we collect the generated code snippets from inference sessions, embed them using `text-embedding-3-large` [36] via OpenAI's API, and cluster the embeddings to identify emergent tool categories. By inspecting and interpreting the resulting clusters, we identify four major classes of tools: (1) **basic image processing**, (2) **advanced image processing**, (3) **visual prompting and sketching**, (4) **numerical and statistical analysis**, and (5) long-tailed operations. We detail each below.

Basic Image Processing. These tools serve as the foundation for visual manipulation and perception. They enable the model to clean, align, and highlight image content in ways that improve downstream reasoning.

- **Cropping**: For high-resolution or cluttered inputs, `PyVision` often crops and zooms into regions of interest. By selecting coordinates through reasoning, it effectively performs soft object detection, focusing attention where it matters most. (Fig. 5)
- **Rotation**: Misaligned images (*e.g.,* rotated maps, skewed documents) can confuse even strong models. `PyVision` rotates inputs to canonical orientations, making text, spatial layouts, or directional cues easier to interpret.
- **Enhancement**: In visually subtle domains like medical imaging, `PyVision` applies contrast adjustments and other enhancements to make latent structures more salient. (Fig. 6)

Advanced Image Processing. These tools reflect `PyVision`'s ability to perform mid to high-level vision tasks, but designed and executed dynamically, on demand.

- **Segmentation**: By isolating specific regions via thresholding or edge detection, `PyVision` can extract foreground objects from background noise.
- **Detection**: `PyVision` generates bounding boxes or edge detection to localize objects in the scene. This supports follow-up operations like counting or measuring. (Fig. 7)
- **OCR**: Without relying on external APIs, `PyVision` extract textual content (*e.g.,* signage, labels) by itself, enabling hybrid visual-linguistic reasoning. (Fig. 5)

Visual Prompting and Sketching. In some tasks, it is not enough to perceive the image—the model must "think visually" [55, 15, 5, 59]. To help itself reason, `PyVision` annotates the image with auxiliary markings, essentially creating visual notes or sketches.

- **Rendering Marks**: In object counting or enumeration task, `PyVision` often marks items with dots or symbols. This external memory acts as a tallying aid, helping it keep track of what's been counted. (Fig. 8)
- **Rendering Lines**: In geometric or spatial tasks (*e.g.,* mazes), `PyVision` draws auxiliary lines to assist reasoning, such as showing the moving directions in a maze.

Numerical and Statistical Analysis. To go beyond perception and into interpretation, `PyVision` invokes tools for quantitative reasoning over visual inputs.

- **Image Histogram**: By plotting pixel intensity distributions, `PyVision` can analyze lighting, contrast, and more, critical for domains where histogram carry meaning. (Fig. 6)
- **Numerical Analysis**: When solving visual math problems or compare quantities, `PyVision` writes scripts to compute areas, lengths, or other metrics for symbolic reasoning. (Fig. 7)

Long-Tail Operations. `PyVision` also invents novel tools not easily classified. These one-off operations showcase its ability to reason creatively under novel constraints. For example, to solve a "spot the difference" task, `PyVision` directly subtracts pixel values between two images and visualizes the result. (Fig. 9) This kind of zero-shot problem decomposition and tool synthesis reflects both the power and flexibility of dynamic tooling for visual reasoning.

Video Reasoning with Agentic Tooling. Video understanding poses unique challenges compared to static image tasks. `PyVision` demonstrates strong potential in this setting by treating video not as a monolithic input but as a sequence of decision points. Rather than exhaustively analyzing all frames, `PyVision` dynamically selects and processes only those frames containing distinct types of tables. (Fig. 10) It then extracts visual evidence and support reasoning. This agentic, multi-step workflow enables `PyVision` to operate more like a human analyst: skimming, sampling, and refining its understanding based on intermediate results.

5.2 Analyzing Tooling Patterns Across Tasks and Domains

Benchmarks. To evaluate the effectiveness of `PyVision` on versatile benchmarks and domains, we select six benchmarks. The details are listed as follows:

- **Multi-Modal Math**: MathVista [26] and MathVision [49] challenge models with math problems that combine visual perception and numerical reasoning.
- **Domain and Logic Reasoning**: MMMU [57] tests subject-specific reasoning across disciplines using multi-modal input, often requiring college-level knowledge. VisualPuzzles [43] focuses on logic, with tasks covering algorithmic, analogical, deductive, inductive, and spatial reasoning, minimizing domain dependency while maximizing abstraction.

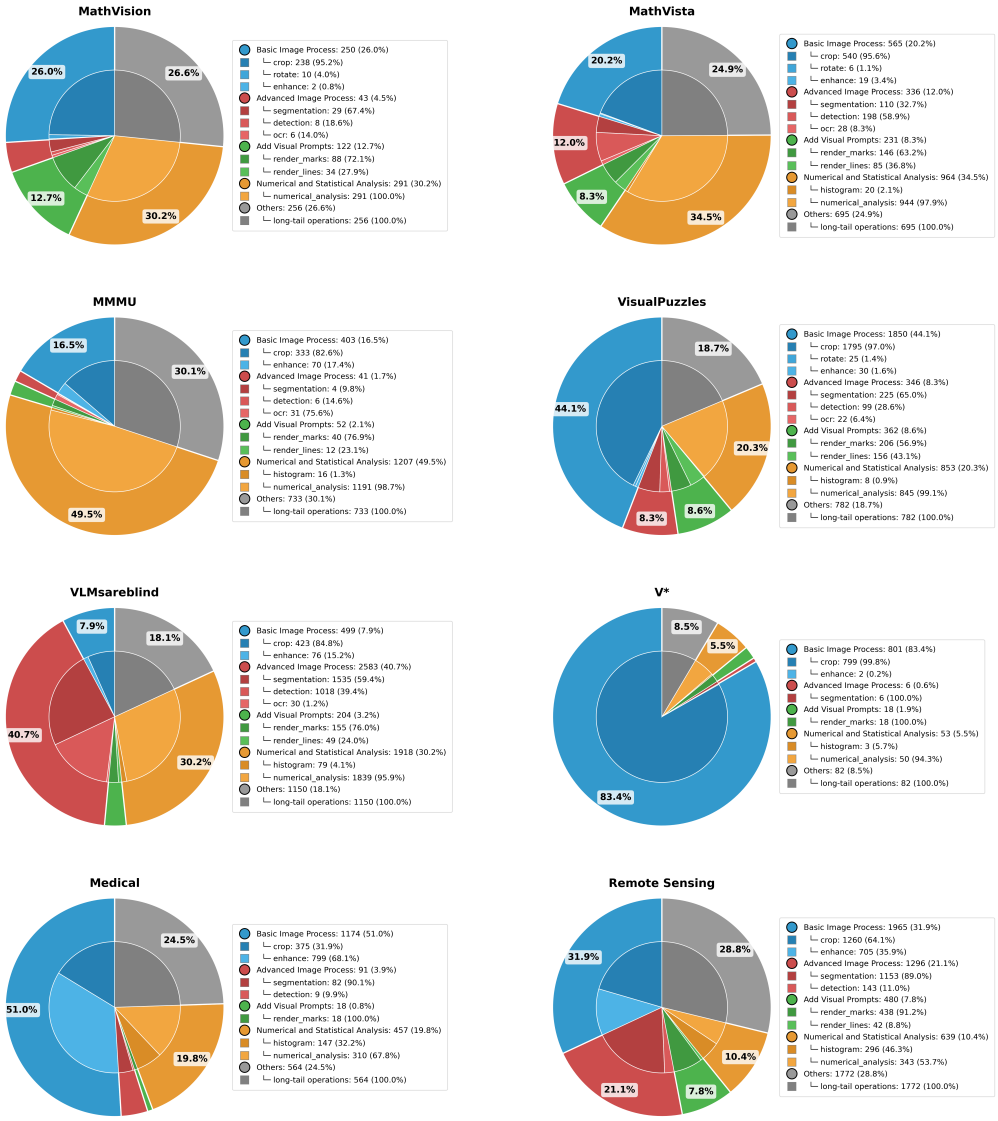

Figure 4: **Taxonomy Distribution Across Benchmarks and Domains.** Tool usage varies significantly across different *tasks* and *domains*.

For math- and logic-related benchmarks, *e.g.,* MathVision [49], MathVista [26], MMMU [57], VisualPuzzles [43], **numerical and statistical** tools constitute a major portion of the usage and **visual prompts** are used relatively more often. In the symbolic vision task VLMsAreBlind [41], **advanced image processing** tools dominate. For visual search in V* [52], `PyVision` primarily relies on **cropping** to facilitate detailed visual querying, which takes over 83% of all tools used.

Tooling preferences are also domain-sensitive: On medical images [17], **contrast-enhancement tools** are frequently invoked. In remote sensing [23], **segmentation tools** are more common.

These observations highlight the importance of flexible and dynamic tooling to support the diverse demands of real-world vision tasks.

- **Symbolic Vision**: VLMs Are Blind [41] consists of designed symbolic visual puzzles, probing the limits of parsing and reasoning over abstract, structured visual primitives.
- **Fine-Grained Visual Search**: V* [52] features 191 high-resolution samples that require pinpointing subtle visual details based on nuanced queries, making it a strong testbed for attention and spatial reasoning.

We also evaluate two special domains, Medical Imaging VQA [17] and Remote Sensing VQA [23] to probe the tooling patterns in different domains.

Distribution of Tools. To understand how `PyVision` adapts its tooling to different problems, we analyze the distribution of tool categories across benchmarks and domains in Fig. 4.

The results reveal strong task- and domain-specific preferences. In math and logic-heavy benchmarks like MathVista [26], MathVision [49], MMMU [57], and VisualPuzzles [43], `PyVision` frequently generates **numerical and statistical** tools to support symbolic and quantitative reasoning. These are often accompanied by **visual prompting and sketching** that help ground abstract logic in visual cues. In symbolic visual tasks such as VLMsAreBlind [41], **advanced image processing** tools are predominant, reflecting the need for structure extraction and visual parsing. For fine-grained visual search tasks like V* [52], **cropping** overwhelmingly dominates, accounting for over 83% of all tools, as the model focuses attention on localized regions.

Domain also plays a significant role: on medical images [17], **contrast enhancement** is commonly used to reveal subtle visual patterns, while in remote sensing [23], **segmentation** tools help delineate objects in large-scale scenes.

These results underscore the importance of dynamic tool generation, allowing the model to flexibly tailor its strategy to the task at hand.

6 Conclusion

We propose `PyVision`, an agentic framework enabling MLLMs to generate and execute Python code on the fly. Different from previous visual programming works [47, 12, 18], `PyVision` needs no visual parsers and predefined static toolset, it generates tools dynamically from the specific query and visual input. We evaluate its effectiveness and flexibility on various benchmarks and visual reasoning scenarios, *e.g.,* medical, multi-modal math problems, remote sensing and visual puzzles. It shows significant performance improvement on versatile benchmarks.

References

[1] M. AI. Kimi k2: Open agentic intelligence, 2025.

[2] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein. Deep compositional question answering with neural module networks. *arXiv:1511.02799*, 2015.

[3] Anthropic. Introducing claude 4, 2025.

[4] A. Backlund and L. Petersson. Vending-bench: A benchmark for long-term coherence of autonomous agents. *arXiv preprint arXiv:2502.15840*, 2025.

[5] H. Bai, Y. Zhou, J. Pan, M. Cemri, A. Suhr, S. Levine, and A. Kumar. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning. *NeurIPS*, 2024.

[6] S. Bai, K. Chen, X. Liu, J. Wang, W. Ge, S. Song, K. Dang, P. Wang, S. Wang, J. Tang, et al. Qwen2.5-vl technical report. *arXiv:2502.13923*, 2025.

[7] G. Bradski. The opencv library. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 2000.

[8] A. Clark et al. Pillow (pil fork) documentation. *readthedocs*, 2015.

[9] J. Feng, S. Huang, X. Qu, G. Zhang, Y. Qin, B. Zhong, C. Jiang, J. Chi, and W. Zhong. Retool: Reinforcement learning for strategic tool use in llms. *arXiv:2504.11536*, 2025.

[10] R. Girshick. The parable of the parser, 2024.

[11] Z. Gou, Z. Shao, Y. Gong, Y. Shen, Y. Yang, M. Huang, N. Duan, and W. Chen. Tora: A tool-integrated reasoning agent for mathematical problem solving. *arXiv preprint arXiv:2309.17452*, 2023.

[12] T. Gupta and A. Kembhavi. Visual programming: Compositional visual reasoning without training. In *CVPR*, 2023.

[13] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al. Array programming with numpy. *Nature*, 2020.

[14] S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[15] W. Hong, W. Wang, Q. Lv, J. Xu, W. Yu, J. Ji, Y. Wang, Z. Wang, Y. Dong, M. Ding, et al. Cogagent: A visual language model for gui agents. In *CVPR*, 2024.

[16] M. Hu, Y. Zhou, W. Fan, Y. Nie, B. Xia, T. Sun, Z. Ye, Z. Jin, Y. Li, Q. Chen, et al. Owl: Optimized workforce learning for general multi-agent assistance in real-world task automation. *arXiv preprint arXiv:2505.23885*, 2025.

[17] Y. Hu, T. Li, Q. Lu, W. Shao, J. He, Y. Qiao, and P. Luo. OmniMedVQA: A new large-scale comprehensive evaluation benchmark for medical lvlm. In *CVPR*, 2024.

[18] Y. Hu, W. Shi, X. Fu, D. Roth, M. Ostendorf, L. Zettlemoyer, N. A. Smith, and R. Krishna. Visual sketchpad: Sketching as a visual chain of thought for multimodal language models. In *NeurIPS*, 2024.

[19] T. Jaeger and K. Klahs. The ebbinghaus illusion: New contextual effects and theoretical considerations. *Perceptual and motor skills*, 2015.

[20] J. Johnson, B. Hariharan, L. van der Maaten, J. Hoffman, L. Fei-Fei, C. L. Zitnick, and R. B. Girshick. Inferring and executing programs for visual reasoning. In *ICCV*, 2017.

[21] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollár, and R. Girshick. Segment anything, 2023.

[22] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa. Large language models are zero-shot reasoners. *NeurIPS*, 2022.

[23] K. Kuckreja, M. S. Danish, M. Naseer, A. Das, S. Khan, and F. S. Khan. Geochat: Grounded large vision-language model for remote sensing. In *CVPR*, 2024.

[24] D. Liu, Z. Wang, M. Ruan, F. Luo, C. Chen, P. Li, and Y. Liu. Visual abstract thinking empowers multimodal reasoning. *arXiv:2505.20164*, 2025.

[25] S. Liu, Z. Zeng, T. Ren, F. Li, H. Zhang, J. Yang, Q. Jiang, C. Li, J. Yang, H. Su, J. Zhu, and L. Zhang. Grounding dino: Marrying dino with grounded pre-training for open-set object detection. In *ECCV*, 2024.

[26] P. Lu, H. Bansal, T. Xia, J. Liu, C. Li, H. Hajishirzi, H. Cheng, K.-W. Chang, M. Galley, and J. Gao. Mathvista: Evaluating mathematical reasoning of foundation models in visual contexts. *arXiv:2310.02255*, 2023.

[27] P. Lu, B. Chen, S. Liu, R. Thapa, J. Boen, and J. Zou. Octotools: An agentic framework with extensible tools for complex reasoning. *arXiv preprint arXiv:2502.11271*, 2025.

[28] M. Luo, N. Jain, J. Singh, S. Tan, A. Patel, Q. Wu, A. Ariyak, C. Cai, S. Z. Tarun Venkat, B. Athiwaratkun, M. Roongta, C. Zhang, L. E. Li, R. A. Popa, K. Sen, and I. Stoica. Deepswe: Training a state-of-the-art coding agent from scratch by scaling rl, 2025. Notion Blog.

[29] MainFunc. Meet genspark super agent, 2025.

[30] Manus. Leave it to manus, 2025.

[31] W. McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for high performance and scientific computing*, 2011.

[32] MiniMax. Minimax-agent, 2025.

[33] OpenAI. Computer-using agent, 2025.

[34] OpenAI. Introducing codex, 2025.

[35] OpenAI. Introducing gpt-4.1 in the api, 2025.

[36] OpenAI. New embedding models and api updates, 2025.

[37] OpenAI. Thinking with images, 2025.

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 2011.

[39] J. Qi, M. Ding, W. Wang, Y. Bai, Q. Lv, W. Hong, B. Xu, L. Hou, J. Li, Y. Dong, and J. Tang. Cogcom: A visual language model with chain-of-manipulations reasoning. In *ICLR*, 2025.

[40] J. Qiu, X. Qi, T. Zhang, X. Juan, J. Guo, Y. Lu, Y. Wang, Z. Yao, Q. Ren, X. Jiang, et al. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*, 2025.

[41] P. Rahmanzadehgervi, L. Bolton, M. R. Taesiri, and A. T. Nguyen. Vision language models are blind. In *ACCV*, 2024.

[42] Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. In *NeurIPS*, 2023.

[43] Y. Song, T. Ou, Y. Kong, Z. Li, G. Neubig, and X. Yue. Visualpuzzles: Decoupling multimodal reasoning evaluation from domain knowledge. *arXiv:2504.10342*, 2025.

[44] A. Su, H. Wang, W. Ren, F. Lin, and W. Chen. Pixel reasoner: Incentivizing pixel-space reasoning with curiosity-driven reinforcement learning, 2025.

[45] Z. Su, L. Li, M. Song, Y. Hao, Z. Yang, J. Zhang, G. Chen, J. Gu, J. Li, X. Qu, et al. Openthinkimg: Learning to think with images via visual tool reinforcement learning. *arXiv preprint arXiv:2505.08617*, 2025.

[46] Z. Su, P. Xia, H. Guo, Z. Liu, Y. Ma, X. Qu, J. Liu, Y. Li, K. Zeng, Z. Yang, et al. Thinking with images for multimodal reasoning: Foundations, methods, and future frontiers. *arXiv preprint arXiv:2506.23918*, 2025.

[47] D. Surís, S. Menon, and C. Vondrick. Vipergpt: Visual inference via python execution for reasoning. In *ICCV*, 2023.

[48] S. Van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu. scikit-image: image processing in python. *PeerJ*, 2014.

[49] K. Wang, J. Pan, W. Shi, Z. Lu, H. Ren, A. Zhou, M. Zhan, and H. Li. Measuring multimodal mathematical reasoning with math-vision dataset. *NeurIPS*, 2024.

[50] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*, 2022.

[51] C. Wu, S. Yin, W. Qi, X. Wang, Z. Tang, and N. Duan. Visual chatgpt: Talking, drawing and editing with visual foundation models. *arXiv:2303.04671*, 2023.

[52] P. Wu and S. Xie. V*: Guided visual search as a core mechanism in multimodal llms. In *CVPR*, 2024.

[53] Z. Xue, L. Zheng, Q. Liu, Y. Li, Z. Ma, and B. An. Simpletir: End-to-end reinforcement learning for multi-turn tool-integrated reasoning, 2025. Notion Blog.

[54] J. Yang, S. Yang, A. Gupta, R. Han, L. Fei-Fei, and S. Xie. Thinking in Space: How Multimodal Large Language Models See, Remember and Recall Spaces. *arXiv:2412.14171*, 2024.

[55] J. Yang, H. Zhang, F. Li, X. Zou, C. Li, and J. Gao. Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v, 2023.

[56] Z. Yang, L. Li, J. Wang, K. Lin, E. Azarnasab, F. Ahmed, Z. Liu, C. Liu, M. Zeng, and L. Wang. Mm-react: Prompting chatgpt for multimodal reasoning and action. *arXiv:2303.11381*, 2023.

[57] X. Yue, Y. Ni, K. Zhang, T. Zheng, R. Liu, G. Zhang, S. Stevens, D. Jiang, W. Ren, Y. Sun, et al. Mmmu: A massive multi-discipline multimodal understanding and reasoning benchmark for expert agi. In *CVPR*, 2024.

[58] X. Zhang, Z. Gao, B. Zhang, P. Li, X. Zhang, Y. Liu, T. Yuan, Y. Wu, Y. Jia, S.-C. Zhu, and Q. Li. Chain-of-focus: Adaptive visual search and zooming for multimodal reasoning via rl, 2025.

[59] B. Zheng, B. Gou, J. Kil, H. Sun, and Y. Su. Gpt-4v(ision) is a generalist web agent, if grounded, 2024.

[60] Z. Zheng, M. Yang, J. Hong, C. Zhao, G. Xu, L. Yang, C. Shen, and X. Yu. Deepeyes: Incentivizing "thinking with images" via reinforcement learning, 2025.

[61] Q. Zhou, R. Zhou, Z. Hu, P. Lu, S. Gao, and Y. Zhang. Image-of-thought prompting for visual reasoning refinement in multimodal large language models. *arXiv:2405.13872*, 2024.

Appendix Contents

A Additional Evaluation Details

A.1 System Prompt Details

System Prompt Template of PyVision

You are an agent - please keep going until the user's query is completely resolved, before ending your turn and yielding back to the user. Only terminate your turn when you are sure that the problem is solved.

Solve the following problem step by step. You now have the ability to selectively write executable Python code to enhance your reasoning process. The Python code will be executed by an external sandbox.

You MUST plan extensively before each function call, and reflect extensively on the outcomes of the previous function calls. DO NOT do this entire process by making function calls only, as this can impair your ability to solve the problem and think insightfully.

For all the provided images, in order, the i-th image has already been read into the global variable `"image_clue_i"` using the `"PIL.Image.open()"` function. When writing Python code, you can directly use these variables without needing to read them again.

Since you are dealing with the vision-related question answering task, you MUST use the python tool (e.g., matplotlib library) to analyze or transform images whenever it could improve your understanding or aid your reasoning. This includes but is not limited to zooming in, rotating, adjusting contrast, computing statistics, or isolating features.

Note that when you use matplotlib to visualize data or further process images, you need to use `"plt.show()"` to display these images; there is no need to save them. Do not use image processing libraries like cv2 or PIL. If you want to check the value of a variable, you MUST use `"print()"` to check it.

The output (wrapped in `"<interpreter>output_str</interpreter>"`) can be returned to aid your reasoning and help you arrive at the final answer. The Python code should be complete scripts, including necessary imports.
Each code snippet is wrapped with:
<code>
```
python code snippet
```
</code>

The last part of your response should be in the following format:
<answer>
`\boxed{"The final answer goes here."}`
</answer>

image resolution:
Image Width: `{width}`; Image Height: `{height}`
user question:
Answer the following Problem with an image provided and put the answer in the format of `\boxed{answer}` `{"query"}`

Remember to place the final answer in the last part using the format:
<answer>
`\boxed{"The final answer goes here."}`
</answer>

A.2 Evaluation Details

Inference Parameters. In the evaluation stage, we set the temperature to 0.6. Here is the chain-of-thought prompt template used for evaluation.

Chain-of-Thought Prompt Template

You are a helpful assistant. And you are dealing with the VQA tasks. Solve the visual questions step by step and give the correct answer. Note: put your answer in the format of `"\boxed{the right answer here}"`
user question: `{query}`

Illustration of GPT-4.1's result in Tab. 2. To keep the consistent evaluation setting with Claude-4.0-Sonnet, we evaluated GPT-4.1 on MathVista [26] and MMMU [57] by ourselves with the above-mentioned CoT prompt.

Illustration of Qwen2.5-VL-72B's result in Sec. 4. When evaluating Qwen2.5-VL-72B on V* [52] and MathVision-testmini [49], to keep the consistent evaluation setting with other models, *e.g.,* GPT-4.1 and Claude-4.0-Sonnet, we wrap the visual question with the above-mentioned CoT prompt template as input. Since Qwen2.5-VL-72B was not specifically optimized for CoT reasoning [6], the results with CoT prompt may be damaged.

B Examples of Generated Tools

B.1 Code Snippet of CROP Tool

```
# The board is towards the center-right.
# Based on the apparent position, let's crop the region centered at (
    center-right), roughly:
# x: 900-1050, y: 520-700 (after visually inspecting the source image
    proportions)
cropped = image_clue_0.crop((900, 520, 1050, 700))

plt.imshow(cropped)
plt.title('Zoom on Advertising Board')
plt.axis('off')
plt.show()
```

Listing 1: **A code snippet example of generated CROP tool.** In this example, `PyVision` was queried to answer the type of the shop based on the advertising board's information. It generated CROP tool to focus on the board area.

B.2 Code Snippet of ROTATE Tool

```
import matplotlib.pyplot as plt
import pytesseract
from PIL import Image

# Step 1: Rotate the image 180 degrees to correct orientation
rotated_image = image_clue_0.rotate(180)

# Display the rotated image to verify orientation
plt.imshow(rotated_image)
plt.axis('off')
plt.show()
```

Listing 2: **A code snippet example of generated ROTATE tool.** In this example, `PyVision` was queried to answer what is written on an upside-down image. It generated ROTATE tool to make the image rotate 180 degree.

B.3 Code Snippet of ENHANCE CONTRAST Tool

```
#Visualize the Image - Display the original image and enhance contrast
    to check for subtle abnormalities.
```

```python
#Let's write code to visualize and enhance the image to better inspect
    for abnormalities.

def enhance_contrast(img):
    arr = np.asarray(img)
    enhanced = np.zeros_like(arr)
    for i in range(3): # For R, G, B channels
        channel = arr[..., i]
        hist, bins = np.histogram(
            channel.flatten(), 256, [0,256])
        cdf = hist.cumsum()
        cdf_normalized = (cdf * hist.max() / cdf.max())
        cdf_m = np.ma.masked_equal(cdf, 0)
        cdf_m = ((cdf_m - cdf_m.min())*255/
                 (cdf_m.max()-cdf_m.min()))
        cdf = (np.ma.filled(cdf_m,0).astype('uint8'))
        enhanced[..., i] = cdf[channel]
    return enhanced
```

Listing 3: **A code snippet example of the generated ENHANCE CONTRAST tool.** In this example, PyVision was queried to identify retinal abnormalities in a fundus image. It generated the ENHANCE CONTRAST tool to increase image contrast, making subtle pathological features more visible.

B.4 Code Snippet of SEGMENTATION Tool

```python
import matplotlib.pyplot as plt
import numpy as np
from skimage.filters import threshold_otsu
from scipy.ndimage import label, binary_opening, binary_closing

# Convert to grayscale
image = image_clue_0
gray = np.array(image.convert('L'))

# Otsu's thresholding on grayscale
otsu_thresh = threshold_otsu(gray)
binary_otsu = gray > otsu_thresh

# Let's apply morphological operations and count the number of
    buildings.

# Morphological operations to clean up the mask
cleaned = binary_opening(binary_otsu, structure=np.ones((5,5)))
cleaned = binary_closing(cleaned, structure=np.ones((7,7)))

# Label connected components
labeled, num_features = label(cleaned)

print("Number of detected buildings:", num_features)

# Calculate area of each region
region_sizes = np.bincount(labeled.flatten())
min_building_area = 1000  # Adjust as needed for urban scale

# Ignore the background (label 0)
num_buildings = np.sum(region_sizes[1:] > min_building_area)

print("Number of buildings (filtered by area):", num_buildings)
```

Listing 4: **A code snippet example of generated SEGMENTATION tool.** In this example, PyVision was queried to count the number of buildings in a satelite remote sensing image. The entire python tool chain (merged into a single python code snippet for conciseness) generated by PyVisionachieved a similar functionality as segmentation models by thresholding, clustering and filtering step by step.

B.5 Code Snippet of DETECTION Tool

```
# The table is roughly at the center, estimate the coordinates to crop
# Image size: 1920x930, table is roughly between (700, 250) and (1200,
    700)
left, top, right, bottom = 700, 250, 1200, 700
table_area = image_clue_0.crop((left, top, right, bottom))
```

Listing 5: **A code snippet example of generated DETECTION tool.** In this example, PyVision was queried to recognize a small cup of water in the image. It generated DETECTION tool to help cropping.

B.6 Code Snippet of OCR Tool

```
import easyocr
import numpy as np

# Convert rotated PIL image to numpy array for easyocr
img_np = np.array(rotated_img)

# Initialize EasyOCR reader
reader = easyocr.Reader(['en'])

# Perform OCR
result = reader.readtext(img_np, detail=0)

# Print the recognized text
print(result)
```

Listing 6: **A code snippet example of generated OCR tool.** In this example, PyVision was queried to answer what is the text on the image. It generated OCR tool , using EasyOCR to recognize the text on the image.

B.7 Code Snippet of RENDER MARKS Tool

```
# Based on my visual analysis, I can identify the following people in
    solid red shirts:
# Let me mark them with circles and numbers

# Person 1: Far left, standing person in bright red shirt
plt.plot(45, 140, 'yo', markersize=15, markeredgecolor='yellow',
    markeredgewidth=3)
plt.text(45, 120, '1', fontsize=16, color='yellow', fontweight='bold',
     ha='center')

# Person 2: Left side, sitting person in red top
plt.plot(110, 195, 'yo', markersize=15, markeredgecolor='yellow',
    markeredgewidth=3)
plt.text(110, 175, '2', fontsize=16, color='yellow', fontweight='bold'
    , ha='center')

# Person 3: Center-right area, person in red shirt sitting
plt.plot(380, 200, 'yo', markersize=15, markeredgecolor='yellow',
    markeredgewidth=3)
plt.text(380, 180, '3', fontsize=16, color='yellow', fontweight='bold'
    , ha='center')

plt.axis('off')
plt.show()
```

Listing 7: **A code snippet example of generated RENDER MARKS tool.** In this example, PyVision was queried to count the people wearing solid red shirts. It generated a RENDER MARKS tool, labeling the people wearing red shirts.

B.8 Code Snippet of RENDER AUXILIARY LINES Tool

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# Load image
img = image_clue_0

# Set up plot
fig, ax = plt.subplots(figsize=(8, 6))
ax.imshow(img)

# Green square location (col, row): (2, 4) in 0-based grid (counting
    from top left)
# Move left 1 step: (1, 4)
start_col, start_row = 2, 4
end_col, end_row = start_col - 1, start_row

# Convert to image coordinates
# Each cell: width = 640/5 = 128, height = 480/5 = 96
cell_w, cell_h = 128, 96

# Center of starting cell
start_x = start_col * cell_w + cell_w // 2
start_y = start_row * cell_h + cell_h // 2
# Center of ending cell
end_x = end_col * cell_w + cell_w // 2
end_y = end_row * cell_h + cell_h // 2

# Draw arrow from green square to left
ax.arrow(start_x, start_y, end_x - start_x, end_y - start_y,
         head_width=20, head_length=20, fc='red', ec='red',
             length_includes_head=True)

# Add annotation at end point
ax.text(end_x, end_y, 'HERE', color='red', fontsize=16, fontweight='
    bold', ha='center', va='center')

plt.axis('off')
plt.show()
```

Listing 8: **A code snippet example of generated RENDER AUXILIARY LINES tool.** In this example, PyVision was queried to answer what is the position after the next step. It generated RENDER AUXILIARY LINES tool to help grounding the position via sketching.

B.9 Code Snippet of VISULIZE IMAGE HISTOGRAM Tool

```
# Plot the histogram of pixel values
img_array = np.array(image_clue_0)
plt.figure(figsize=(6,4))
plt.hist(img_array.ravel(), bins=256, color='gray')
plt.title('Pixel Value Histogram')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.show()
```

Listing 9: **A code snippet example of generated VISULIZE IMAGE HISTOGRAM tool.** In this example, PyVision was queried to if there is some specific pattern. It generated VISULIZE IMAGE HISTOGRAM tool to help detect the pattern.

B.10 Code Snippet of NUMERICAL ANALYSIS Tool

```
import numpy as np

# Convert the glass area to a numpy array
glass_np = np.array(glass_area)

# Calculate mean RGB values
mean_rgb = glass_np.mean(axis=(0, 1))
print("Mean RGB values of the glass area:", mean_rgb)
```

Listing 10: **A code snippet example of generated NUMERICAL ANALYSIS tool.** In this example, PyVision was queried to answer the the color of the specific area in the image. It generated NUMERICAL ANALYSIS tool to help analysis the color.

C Case Study of PyVision

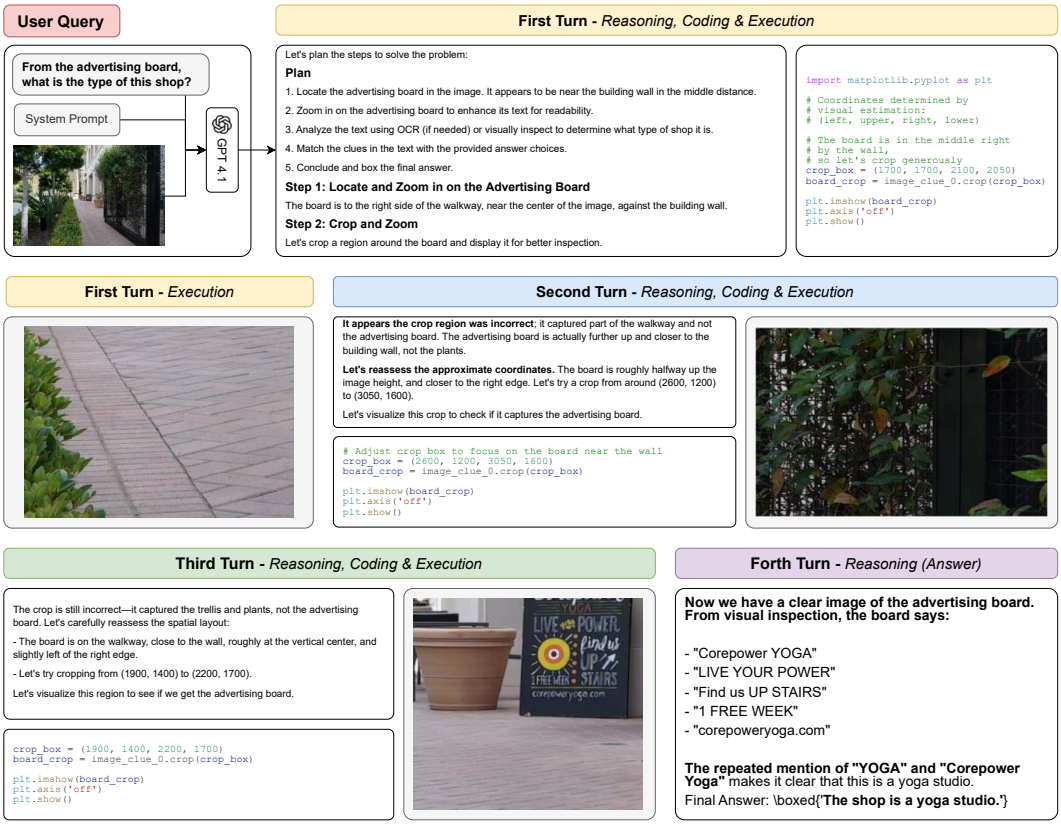

Figure 5: **Case Study: Visual Search** requires language-based world knowledge, vision-based contextual understanding, and iterative refinement to precisely locate specific visual elements. In this case from V* [52], the agent must identify text on a small advertising board in a complex outdoor scene. The target occupies a minimal image area, requiring adaptive cropping. This showcases how dynamic tool generation enables flexible spatial reasoning.

The agent generates custom Python codes across three iterative turns, refining coordinate-based **cropping tools** based on visual feedback, and eventually isolating the advertising board required by the user query. The agent then performs **OCR** and extracts key text such as "YOGA" and correctly identifies the business as a yoga studio.

[†]Data source: https://www.jabobaby.com/blog/posts/photo-hunt

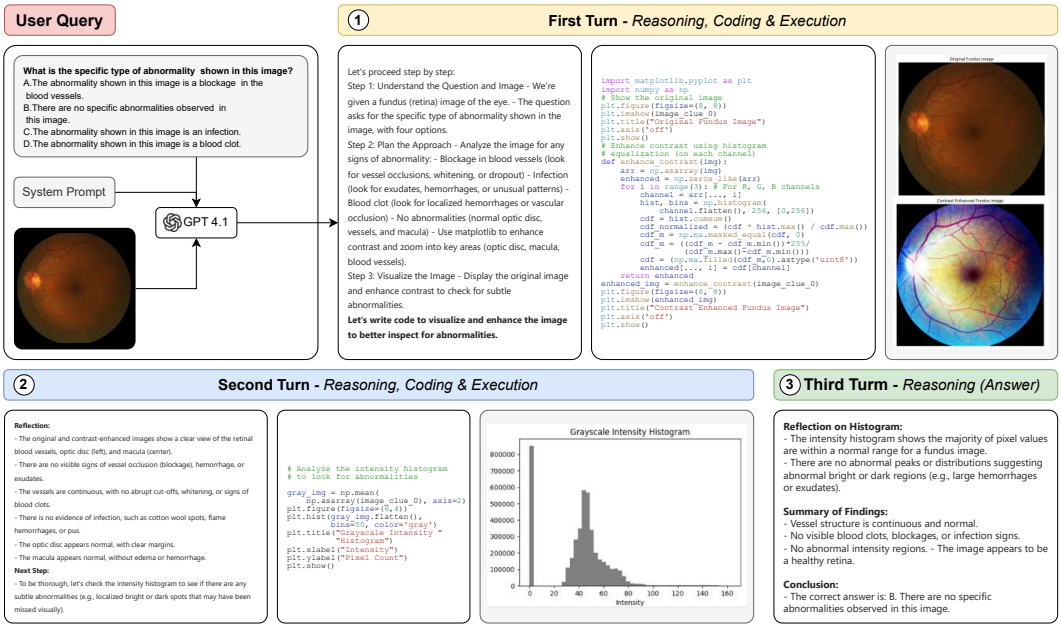

Figure 6: **Case Study: Medical Imaging Analysis** often requires high visual sensitivity and domain-specific reasoning to solve subtle medical problems. This case from OmniMedVQA [17] involves identifying a specific abnormality in a fundus image of the retina. `PyVision` is tasked with diagnosing the image without prior domain-specific modules.

To begin, `PyVision` generates a visualization tool that applies histogram equalization to **enhance contrast** in key regions, a standard technique in medical imaging analysis. It then creates a second tool to plot the **intensity histogram**, confirming the absence of abnormal peaks. Within a few iterations, `PyVision` constructs a custom diagnostic pipeline from scratch, correctly concluding that no specific abnormalities are present. This highlights `PyVision`'s ability to generate interpretable, domain-adapted tools for complex medical tasks.

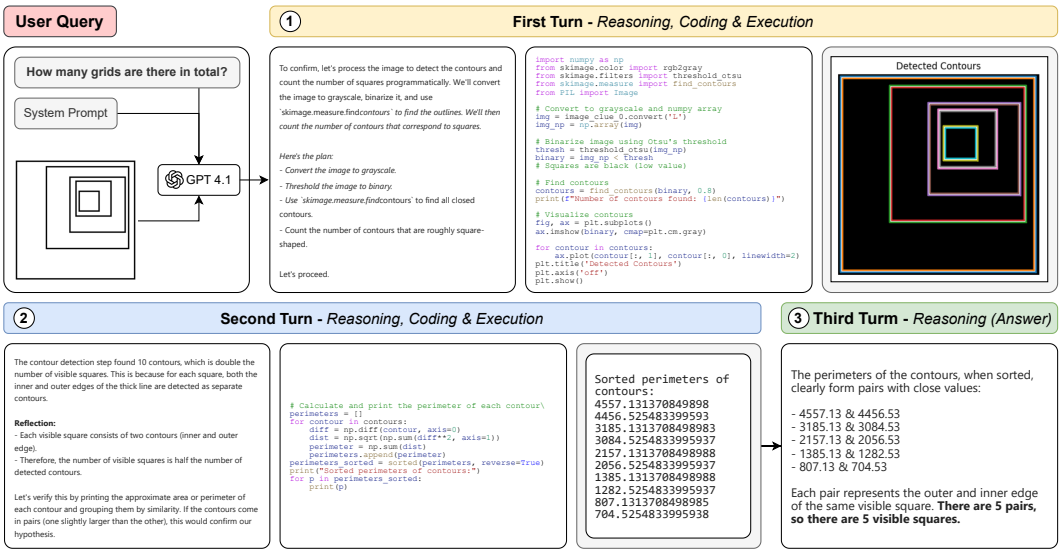

Figure 7: **Case Study: Symbolic Visual Puzzles** require precise spatial reasoning and the recognition of overlapping or closely positioned geometric primitives, a task that is simple for humans, yet surprisingly difficult for powerful MLLMs when asked directly [41]. In contrast, the regular structure of such puzzles makes them well-suited to code-based solutions.

This is a nested squares counting task from VLMsAreBlind [41]. First, `PyVision` applies **edge detection** using `skimage.measure.find_contours` and identifies ten contours. Then, it infers that the number of contours corresponds to five nested squares, as each square contributes an inner and outer edge. To validate this, the model performs **numerical analysis** and prints the sorted perimeters of the detected contours. In the third stage, it performs a double-check and confidently confirms the correct answer: five nested squares.

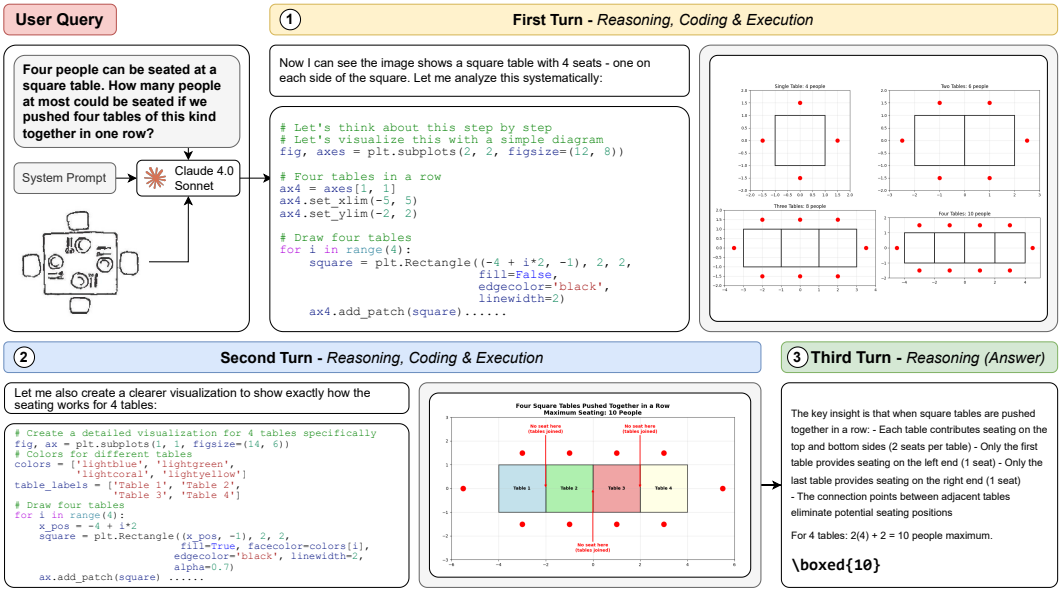

Figure 8: **Case Study: Visual Sketching** is a valuable strategy for humans to solve scientific problems, including those in mathematics and physics. It can also enhance AI model performance by enabling precise numerical calculations and visual reasoning [18].

In this example from MathVision [49], `PyVision` is asked to compute how many people can sit around a row of four tables. The model first sketches the table arrangement, **rendering marks** to represent four connected tables and people sitting around them. From the sketch, it infers that ten people can be seated. In a second turn, `PyVision` generates a schematic diagram to explain and validate the seating logic, identifying positions that are unsuitable for seating.

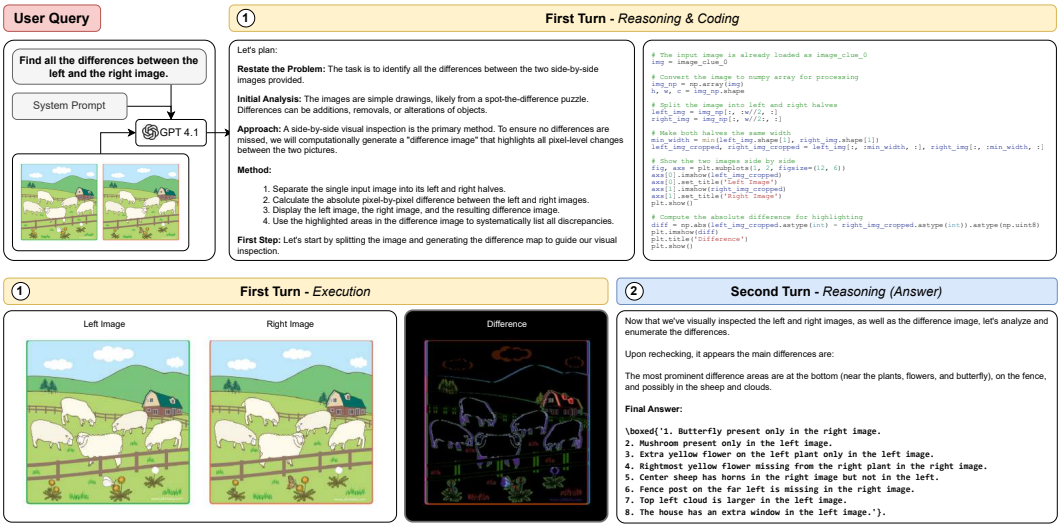

Figure 9: **Case Study: Spot-the-Difference** showcases structured visual comparison. Given a side-by-side image pair, `PyVision` is asked to identify all visual discrepancies.

`PyVision` first plans a multi-step strategy: it splits the image into left and right halves, computes the absolute pixel-level difference, and generates a difference map to highlight changes. It then displays both original images alongside the computed difference visualization to aid analysis. Based on this, `PyVision` proceeds to enumerate the identified differences. Although the final answers are *not completely correct*, the model's initiative to employ pixel-level differencing and organize a reasoning pipeline is notable. This example illustrates both the creative potential of agentic visual reasoning and the ongoing challenge of mitigating hallucinations.†

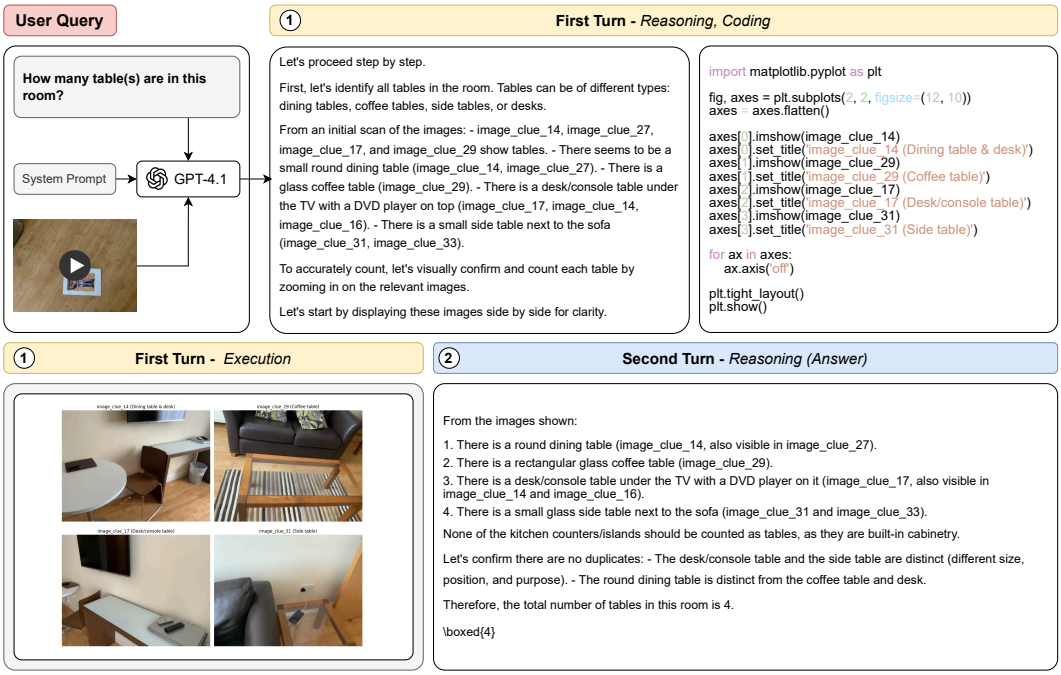

Figure 10: **Case Study: Video Understanding** benefits from an agentic pipeline that integrates reasoning across frames and targeted frame selection. In this example from VSI-Bench [54], a 3-minute egocentric video of an indoor scene is presented, and `PyVision` is tasked with identifying the number of tables in the room.

`PyVision` begins by analyzing the video to **detect** candidate frames containing tables. It then selects and displays key frames, each showing a different table, including dining table, desk, coffee table, and side table, to support its reasoning. By synthesizing visual evidence and textual inference across multiple views, `PyVision` concludes there are four distinct tables in the room.