MuSEAgent: A Multimodal Reasoning Agent with Stateful Experiences

Anonymous ACL submission

Abstract

Multimodal agents have shown growing ability to interleave tool use with multimodal observations for complex reasoning, yet effectively reusing past interactions remains challenging. In this paper, we introduce **MuSEAgent**, a multimodal reasoning agent that enhances decision-making by discovering and leveraging stateful experiences. Rather than relying on trajectory-level retrieval, we propose a stateful experience learning paradigm that abstracts interaction data into atomic decision experiences through hindsight reasoning. These experiences are organized into a quality-filtered experience bank that supports stateful experience exploitation at inference time. Specifically, MuSEAgent performs a Deep-and-Wide experience search across compositional semantic viewpoints. Extensive experiments show that MuSEAgent seamlessly improves base models with state-level guidance on complex multimodal reasoning tasks. The gains hold even when the agent itself acts as the hindsight reasoner, and stateful experiences transfer across datasets without any in-domain interaction, still surpassing in-domain trajectory-level baselines.

1 Introduction

Multimodal agents have achieved significant progress in perception and generation across heterogeneous modalities (He et al., 2024; Xie et al., 2024; Yang et al., 2023). A core challenge facing these agents is learning to effectively exploit tools in order to navigate complex, multi-step environments. A promising direction is to retrieve and learn from similar past experiences stored in a memory bank (Packer et al., 2023; Shinn et al., 2023; Zhao et al., 2024). Recent work further explores experience-augmented agents that leverage historical interaction trajectories to improve autonomy and generalization (Zhao et al., 2024; Shinn et al., 2023; Wang et al., 2023). However, extending these ideas to multimodal agents introduces two fundamental challenges. First, visual inputs typically carry much lower information density than textual ones, such that retrieving entire interaction trajectories often injects redundant or irrelevant context, amplifying reasoning noise under constrained context windows (Liu et al., 2024; Chang et al., 2024). Second, multimodal reasoning requires interleaved thinking across diverse modalities, which makes identifying relevant and similar memory cases substantially more difficult. Moreover, trajectory-level experience retrieval fails to provide fine-grained decision guidance when agents encounter intermediate reasoning bottlenecks, where state-specific tactical knowledge is far more beneficial than coarse task-level analogies (Zhang et al., 2025; Xie et al., 2024; Zhou et al., 2025).

In this paper, we propose MuSEAgent, a novel **Mu**ltimodal Reasoning Agent with **S**tateful **E**xperiences. Specifically, we reformulate multimodal agent reasoning as a stateful experience learning process, in which historical trajectories are abstracted into atomic state-action pairs through hindsight reasoning. These abstracted experiences, termed stateful experiences, are organized into a quality-filtered experience bank that supports retrieval conditioned on the agent's current decision state, rather than on static initial task contexts, enabling more precise and noise-free guidance at each reasoning step.

To effectively exploit stateful experiences during reasoning, MuSEAgent extends its reasoning capability with multi-round retrieval that iteratively queries the experience bank during inference. Specifically, we design a compositional state representation mechanism that decomposes complex multimodal states into multiple semantic viewpoints, enabling experience indexing across perceptual intent, tool execution history, and interaction context. Based on this representation,

the agent performs *Wide Search* to retrieve cross-task strategic knowledge and *Deep Search* to iteratively refine retrieval across complementary semantic viewpoints within a single reasoning step.

We evaluate MuSEAgent on four multimodal benchmarks across three base models. Experimental results demonstrate that our approach seamlessly improves base models by incorporating state-level experience retrieval, outperforming strong trajectory-level baselines by nearly 8% in average accuracy. The gains are especially pronounced on complex multimodal reasoning, where state-level guidance mitigates contextual noise. These improvements hold even when the agent itself acts as the hindsight reasoner, and stateful experiences transfer across datasets without any in-domain interaction, still surpassing in-domain trajectory-level baselines.

2 Related Work

Multimodal Agent. Multimodal agents shift visual reasoning from single-pass inference to interactive tool-using decision-making (Xi et al., 2025; Wang et al., 2024). Inspired by ReAct (Yao et al., 2022), prior works coordinate vision modules or structured programs for complex visual reasoning (Yang et al., 2023; Gupta and Kembhavi, 2023; Surís et al., 2023; Shen et al., 2023), but typically append entire interaction histories or follow rigid execution trajectories (He et al., 2024; Zhang et al., 2025), introducing contextual redundancy and undermining long-horizon consistency. Our MuSEAgent instead formalizes agentic multimodal reasoning as a Stateful Experience-Based Markov Decision Process (MDP), converting interaction histories into fine-grained, state-level experiences.

Experience-driven Agent Learning. Recent work explores memory-augmented agents that reuse past trajectories (Shinn et al., 2023; Wang et al., 2023; Zhao et al., 2024; Packer et al., 2023), with most retrieving experiences at the coarse trajectory level (Yao et al., 2023; Zhu et al., 2023), where long, rigid trajectories inject noise and provide only task-level analogies that ill-suit fine-grained multimodal reasoning (Yao et al., 2022; Yang et al., 2023). A complementary line formalizes agent memory in MDP-style frameworks with value-guided case selection (Zhou et al., 2025); however, applying such formulations to long-horizon multimodal reasoning faces fundamental challenges: state-level rewards are unavailable, and credit assignment from trajectory-end signals is unreliable, leading existing instantiations (Zhou et al., 2025) to retain only trajectory-end transitions in the memory bank. To address these challenges, MuSEAgent models multimodal reasoning as a state-level decision process and adopts hindsight-distilled natural-language guidance, in place of scalar value, as the supervision signal, learning fine-grained decision guidance attached to each transition.

3 MuSEAgent

We present MuSEAgent (Figure 1), a novel framework that enhances multimodal agent reasoning by abstracting and exploiting stateful experiences. MuSEAgent operates as a multi-step tool-using agent that iteratively invokes multimodal tools and updates its execution history at each reasoning step. We formulate this process as a Stateful Experience-Based MDP (Sec. 3.1), abstract trajectories into reusable state-level experiences via hindsight reasoning (Sec. 3.2), and exploit them online through a Deep-and-Wide search (Sec. 3.3).

3.1 Problem Formulation

We formally characterize the sequential decision-making process of MuSEAgent as a Stateful Experience-Based MDP, in which new problems are solved by retrieving fine-grained experiences attached to similar decision states, rather than entire historical trajectories.

Definition 3.1 (Stateful Experience-Based MDP)**.** Let $u \in \mathcal{U}$ denote the natural language query provided at the beginning. The agent interacts with multimodal observations and historical execution contexts across multiple reasoning steps. The decision state at step t is defined as $s_t = (u, v_t, d, H_t)$, where u is the fixed user instruction, v_t denotes the current visual observation or perceptual input, d is an optional task descriptor, and H_t is the execution history up to step t. The action space is defined as $a_t \in \mathcal{A}_{\text{tool}} \cup \mathcal{A}_{\text{exp}}$, where $\mathcal{A}_{\text{tool}}$ contains task-specific execution tools and $\mathcal{A}_{\text{exp}}$ corresponds to experience retrieval over an experience bank. State transitions are stochastic in general, since tool outputs and visual observations may be noisy or non-deterministic. A trajectory generated by the agent is represented as $\tau = (s_1, a_1, s_2, a_2, \ldots, s_T, a_T)$, where T denotes the task-dependent reasoning horizon. The trajectory outcome $o \in \{0, 1\}$ indicates whether the

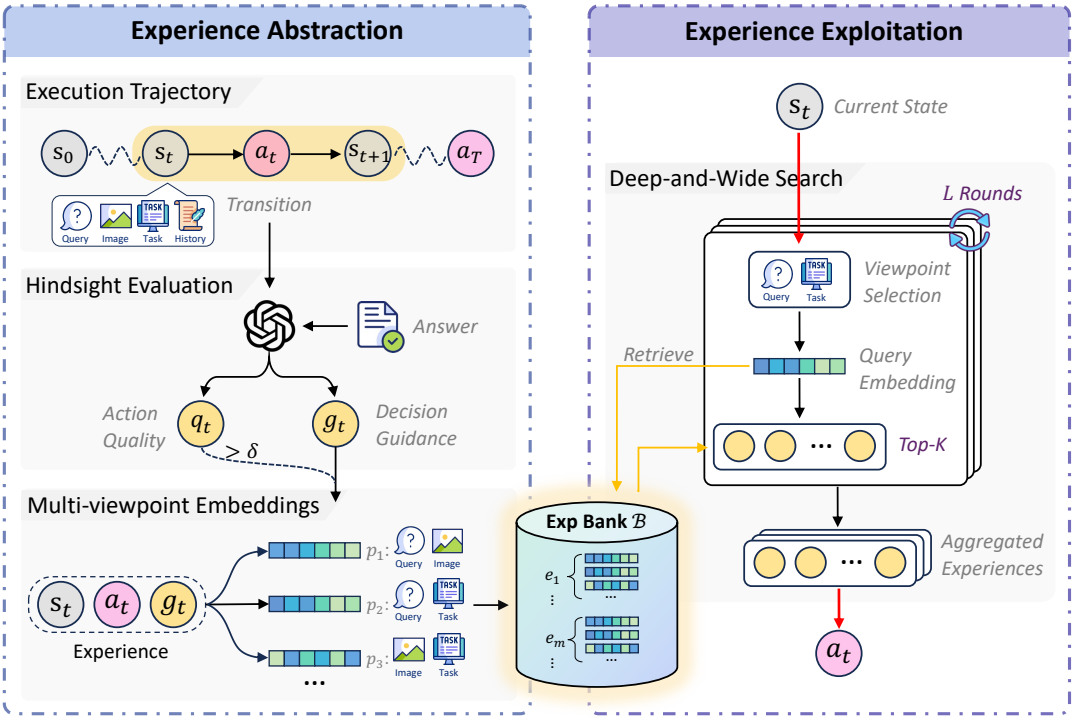


Figure 1: The overview of MuSEAgent. The framework consists of two phases: (1) **Experience Abstraction**, which extracts state-level experiences via hindsight evaluation and builds multi-viewpoint embeddings for each experience; (2) **Experience Exploitation**, where the agent performs a Deep-and-Wide search over the experience bank to determine the next action.

final answer produced at s_T matches the ground truth. Each transition serves as an experience unit.

Unlike prior MDP-style memory frameworks (Zhou et al., 2025) that rely on scalar value to ground action quality, this formulation does not assume state-level rewards, since they are unavailable in long-horizon multimodal reasoning and credit assignment from trajectory-end signals is ambiguous. We instead supervise each experience unit with hindsight-distilled natural-language guidance, as detailed in Sec. 3.2.

3.2 Stateful Experience Abstraction via Hindsight Reasoning

Experience Abstraction. To construct a compact and noise-resistant experience bank, we first decompose agent trajectories into atomic state–action transitions (s_t, a_t, s_{t+1}). These transitions serve as candidate experience units that capture localized decision contexts within long multimodal interaction trajectories, including steps where tools return erroneous or unexpected outputs, both informative for hindsight reasoning. Each transition is then evaluated using a hindsight reasoning model that assesses the quality of the decision and extracts reusable decision guidance. Specifically, we instantiate $\mathcal{Q}_\phi$ with a multimodal reasoning model that, conditioned on the realized outcome of the trajectory, takes the transition context as input and produces both a scalar quality score q_t and a textual guidance summary g_t:

$$(q_t, g_t) = \mathcal{Q}_\phi(s_t, a_t, s_{t+1} \mid \tau, o), \qquad (1)$$

where $q_t \in [0, 10]$ reflects the estimated decision quality of the action taken at state s_t, and g_t summarizes the key decision experience abstracted from this transition.

To suppress noisy or uninformative transitions, we retain only high-quality evaluated transitions to construct the experience bank:

$$\mathcal{B} = \{e_t = (s_t, a_t, g_t) \mid q_t \geq \delta\}, \qquad (2)$$

where δ is a predefined quality threshold (default $\delta = 5.0$). Each retained transition is thus converted into a reusable experience unit consisting of the decision context (s_t), the executed action (a_t), and the abstracted decision guidance (g_t).

Compositional State Representation. Multimodal agent states are heterogeneous, including textual instructions, visual observations, task descriptors, and execution histories, which provide

complementary signals for experience retrieval. To enable flexible querying over such heterogeneous states, we organize each state into multiple semantic viewpoints. Let $\mathcal{P} = \{p_1, p_2, \ldots, p_M\}$ enumerate all $M = 15$ non-empty compositions of the four state components (u, v_t, d, H_t) (e.g., $p_i(s_t) = (u, v_t)$ for perceptual intent, $p_i(s_t) = (d, H_t)$ for tool execution history, or $p_i(s_t) = (u, v_t, d, H_t)$ for the full task context). During the offline experience indexing stage, each experience e_t is associated with multiple embeddings derived from the corresponding state components under each viewpoint:

$$\mathbf{z}_t^{(i)} = f_\theta(p_i(s_t)), \quad p_i \in \mathcal{P}, \tag{3}$$

where f_θ denotes a multimodal embedding model and $p_i(s_t)$ extracts the state elements for viewpoint p_i. As a result, each experience e_t is indexed under multiple complementary semantic viewpoints, enabling the agent to retrieve relevant experiences through different contextual cues. At inference time, the agent policy adaptively selects an appropriate viewpoint to query the experience bank.

The complete procedure is summarized in Algorithm 1 of Appendix A.

3.3 Stateful Experience Exploitation via Deep-and-Wide Search

At inference step t, before committing to the next action, the agent consults the experience bank to obtain additional decision guidance. The agent policy $\pi(\cdot \mid s_t)$ first selects a semantic viewpoint $p_i \in \mathcal{P}$ based on the current state s_t. Since each experience e_i in the experience bank $\mathcal{B}$ is indexed with viewpoint-specific embeddings, the selected viewpoint determines which embedding index is queried. The agent then constructs a query embedding and retrieves relevant experiences from the corresponding viewpoint index. To balance generalization and precise state matching, we adopt two complementary retrieval strategies: *Wide Search*, and *Deep Search*.

Wide Search. To identify broadly relevant experiences, Wide Search performs breadth-oriented retrieval under the currently selected viewpoint. Given the query embedding $\mathbf{q}_t^{(i)} = f_\theta(p_i(s_t))$ constructed under the selected viewpoint p_i, the agent retrieves the Top-K most relevant experiences from the corresponding viewpoint index according to cosine similarity:

$$\mathcal{E}_{\text{wide}}(s_t) = \text{Top-K}_{e_m \in \mathcal{B}} \ \text{sim}\left(\mathbf{q}_t^{(i)}, \mathbf{z}_m^{(i)}\right), \tag{4}$$

where $\mathbf{z}_m^{(i)}$ denotes the stored embedding of experience e_m under the selected viewpoint, and K controls the retrieval breadth. By retrieving multiple experiences with similar contextual signals, Wide Search exposes the agent to diverse decision contexts and helps identify reusable reasoning patterns that generalize across related tasks.

Deep Search. When broader retrieval does not provide sufficiently precise guidance, Deep Search performs iterative refinement by querying the experience bank under multiple semantic viewpoints. At each refinement round j, the agent selects a viewpoint $p_{i_j} \in \mathcal{P}$ and constructs a viewpoint-specific query embedding

$$\mathbf{q}_t^{(i_j)} = f_\theta(p_{i_j}(s_t)). \tag{5}$$

The agent then retrieves the most relevant experience under that viewpoint:

$$\mathcal{E}_{\text{deep}}(s_t) = \bigcup_{j=1}^{L} \text{Top-1}_{e_m \in \mathcal{B}} \ \text{sim}\left(\mathbf{q}_t^{(i_j)}, \mathbf{z}_m^{(i_j)}\right), \tag{6}$$

where L denotes the maximum number of refinement rounds. Each round emphasizes a different aspect of the state representation, allowing the agent to progressively align task intent, perceptual observations, and historical tool usage patterns. For example, the agent may initiate retrieval under a purely visual viewpoint to resolve bounding-box ambiguities, and subsequently issue a secondary query under the execution histories (i.e., tool) viewpoint to validate tool parameter syntax, complementing Wide Search.

Unified Deep-and-Wide Search. In practice, experience search is performed iteratively under a sequence of viewpoints selected by the agent. At round j, a viewpoint p_{i_j} is chosen, and the agent retrieves the Top-K most relevant experiences under that viewpoint. The final experience set is

$$\mathcal{E}_{\text{deep-wide}}(s_t) = \bigcup_{j=1}^{L} \text{Top-K}_{e_m \in \mathcal{B}} \ \text{sim}\left(\mathbf{q}_t^{(i_j)}, \mathbf{z}_m^{(i_j)}\right), \tag{7}$$

where L is the number of retrieval rounds and K controls the retrieval breadth per round. The retrieved experience strings $\{g_m\}$ are injected into the agent's working context to decide the next action $a_t \sim \pi(\cdot \mid s_t, \mathcal{E}_{\text{deep-wide}})$.

The complete online retrieval-and-decision procedure is summarized in Algorithm 2 of Appendix A. A representative case illustrating both

Table 1: Statistics of Benchmark Datasets.

Split	V* Bench			MME-RealWorld-Lite			Zoom-Bench	HR-Bench
	Direct Attr.	Rel. Pos.	Total	Percept.	Reas.	Total		
Exploration	57	38	95	584	375	959	310	100
Evaluation	58	38	96	585	375	960	311	100

stateful experience abstraction and Deep-and-Wide exploitation is provided in Appendix G.

4 Experiments

We design our empirical investigation around four aspects: overall performance against trajectory-level baselines (Sec. 4.2), the role of Deep-and-Wide search (Sec. 4.3), cross-dataset transferability (Sec. 4.4), and ablations on key components (Sec. 4.5), each isolating one design dimension.

4.1 Experimental Setup

Benchmark Datasets. We evaluate on four multiple-choice VQA benchmarks that demand fine-grained visual perception, real-world visual understanding, and complex multimodal reasoning: V* Bench (Wu and Xie, 2024), MME-RealWorld-Lite (Zhang et al., 2024), Zoom-Bench (Wei et al., 2026), and HR-Bench (Wang et al., 2025). Following the standard practice in experience-augmented agents (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2023), where the agent first interacts with a portion of the data to accumulate experiences and is then evaluated on the rest, we sample each benchmark into two equally sized halves (Table 1): the first half serves as the *exploration set* where the agent collects interaction trajectories for building the experience bank, and the second half serves as the *evaluation set* on which we report final accuracy. Detailed dataset descriptions are provided in Appendix B.

Baseline Methods. We compare MuSEAgent against four baselines spanning vanilla reasoning, Tool-Integrated Reasoning (TIR) (Lin and Xu, 2025), and trajectory-level experience methods: Vanilla CoT (Kojima et al., 2022) (no tools), ReAct (Yao et al., 2022) (dynamic tool usage), Reflexion (Shinn et al., 2023) (reflections from failed trajectories), and Expel (Zhao et al., 2024) (insights from both successful and failed trajectories). See Appendix C for details.

Implementation Details. We use three agentic MLLMs as base models, Qwen3-VL-32B-Instruct, Qwen3-VL-235B-A22B-Instruct (Bai et al., 2025), and Qwen3.5-397B-A17B (Qwen Team, 2026), with GPT-4o (Hurst et al., 2024) as the default hindsight reasoning model $\mathcal{Q}_\phi$ in Sec. 4.2, and Qwen3-VL-8B-Embedding (Li et al., 2026) as the retrieval encoder. We set $L = 3$, $K = 3$, and $\delta = 5.0$. Prompts and efficiency details are in Appendix H and Appendix E.

Tool Bank. The agent is equipped with 13 multimodal tools spanning extraction and retrieval (OCR, equation solving, calculator, web search), visual processing (localization, zoom-in, crop, region highlighting, region/object depth), and cross-modal alignment (image-image, image-text, text-image similarity). Full specifications are in Appendix D. All baselines share this tool bank.

Experimental Protocol. We evaluate MuSE-Agent under two deployment scenarios. *In-domain experience accumulation* (Sec. 4.2): the agent builds its experience bank from the target benchmark's exploration set; all experience-augmented baselines (Reflexion, Expel, MuSE-Agent) consume the same in-domain exploration set, isolating the contribution of the experience representation and retrieval mechanism. *Cross-dataset experience transfer* (Sec. 4.4): MuSE-Agent operates on a pooled bank from sibling benchmarks held out from the target, with no in-domain interaction, while baselines remain under their original in-domain setting as reference points. This deliberately disadvantages MuSEAgent against the in-domain baselines, testing whether stateful experiences capture reusable, transferable skills rather than target-specific shortcuts. Accuracy is reported on the evaluation set.

4.2 Overall Performance

Table 2 presents the main results across various methods and base models. From these results, we derive two key findings:

Table 2: Performance comparison of MuSEAgent against baselines on various benchmarks under three base models. We report the accuracy (%) for all tasks. Best results are highlighted in **bold**.

Method	Exp. Level	V* Bench			MME-RealWorld-Lite			Zoom-Bench	HR-Bench	Avg.
		Direct Attr.	Rel. Pos.	Micro Avg.	Percep.	Reas.	Micro Avg.			
Qwen3-VL-32B-Instruct										
Vanilla CoT (Kojima et al., 2022)	N/A	27.59	57.89	39.58	28.55	28.53	28.54	31.83	51.00	37.74
w/ ReAct (Yao et al., 2022)	N/A	62.07	84.21	70.83	44.44	36.80	41.46	44.69	66.00	55.75
w/ Reflexion (Shinn et al., 2023)	Trajectory	74.13	76.32	75.00	46.15	34.13	41.46	48.87	64.00	57.33
w/ Expel (Zhao et al., 2024)	Trajectory	70.69	78.95	73.96	42.39	**40.27**	41.56	46.30	66.00	56.96
w/ MuSEAgent	State	**86.21**	**84.21**	**85.42**	**46.32**	38.13	**43.13**	**54.66**	**78.00**	**65.30**
Qwen3-VL-235B-A22B-Instruct										
Vanilla CoT (Kojima et al., 2022)	N/A	24.14	47.37	33.33	33.68	29.87	32.19	36.98	54.00	39.13
w/ ReAct (Yao et al., 2022)	N/A	44.83	68.42	54.17	42.74	35.20	39.79	43.73	63.00	50.17
w/ Reflexion (Shinn et al., 2023)	Trajectory	48.28	86.84	63.54	43.42	31.73	38.85	42.44	69.00	53.46
w/ Expel (Zhao et al., 2024)	Trajectory	53.45	73.68	61.46	42.39	34.67	39.38	43.09	72.00	53.98
w/ MuSEAgent	State	**55.17**	**92.11**	**69.79**	**42.91**	**35.73**	**40.10**	**45.02**	**74.00**	**57.23**
Qwen3.5-397B-A17B										
Vanilla CoT (Kojima et al., 2022)	N/A	39.66	60.53	47.92	39.32	36.80	38.33	39.23	61.00	46.62
w/ ReAct (Yao et al., 2022)	N/A	79.31	71.05	76.04	50.26	44.00	47.81	59.81	75.00	64.67
w/ Reflexion (Shinn et al., 2023)	Trajectory	77.59	73.68	76.04	51.97	43.47	48.65	61.74	72.00	64.61
w/ Expel (Zhao et al., 2024)	Trajectory	81.03	76.32	79.17	53.68	42.93	49.48	59.16	76.00	65.95
w/ MuSEAgent	State	**84.48**	**78.95**	**82.29**	**54.87**	**46.40**	**51.56**	**67.20**	**78.00**	**69.76**

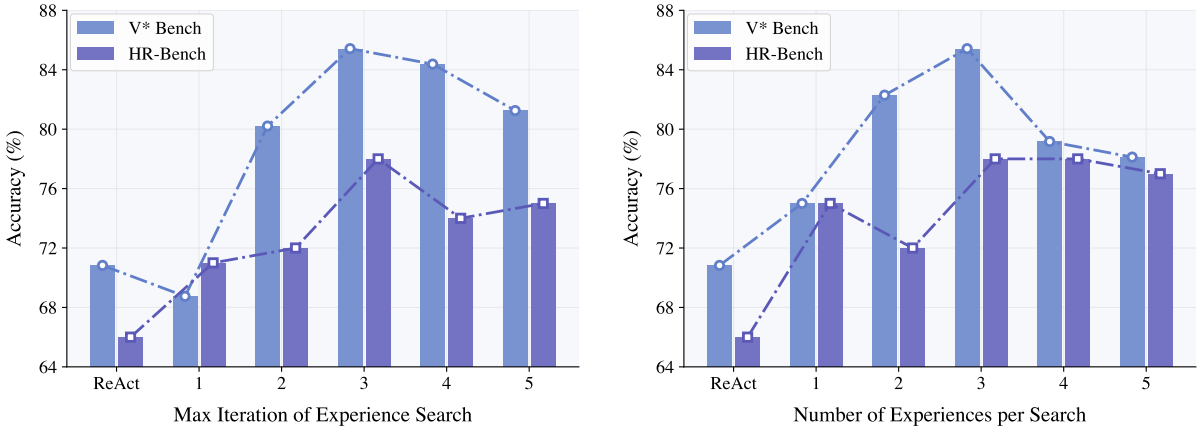
Stateful experiences consistently outperform trajectory-level baselines. MuSEAgent achieves the highest average accuracy across all evaluated benchmarks and base models. Compared with trajectory-level methods such as Reflexion and Expel, MuSEAgent shows substantial gains. For example, on Qwen3-VL-32B-Instruct, MuSEAgent reaches 65.30% average accuracy, surpassing the strongest baseline by nearly 8%. On the fine-grained V* Bench Relative Position task with Qwen3-VL-235B-A22B-Instruct, our method improves accuracy by 18.43% over Expel. These results suggest that decoupling historical episodes into granular state-action pairs mitigates the noise of monolithic trajectories. By retrieving precise stateful experiences, the agent resolves multimodal reasoning bottlenecks more effectively than agents that rely on coarse task-level analogies, even under matched exploration data.

Stateful-experience gains hold across base-model capacities. MuSEAgent improves the strongest trajectory-level baseline across all three base models, by +7.97, +3.25, and +3.81 points on Qwen3-VL-32B-Instruct, Qwen3-VL-235B-A22B-Instruct, and Qwen3.5-397B-A17B respectively, while absolute accuracy continues to grow with model size, reaching 69.76% on Qwen3.5-397B-A17B. The fact that a 397B-parameter model still benefits from state-level guidance indicates that stateful experiences provide information that is complementary to scale rather than substitutable by it, across two distinct model families.

4.3 Investigation on Deep-and-Wide Experience Search

To probe how depth and breadth shape stateful retrieval, we scale one dimension at a time on Qwen3-VL-32B-Instruct while fixing the other at 3. As Figure 2 shows, both dimensions yield substantial gains over the zero-experience ReAct baseline and follow a non-monotonic trend: accuracy rises sharply from 0 to 3, peaks at 3 (85% on V* Bench, 78% on HR-Bench), and then slightly declines at 4 or 5, while every configuration still beats the experience-free baseline. Wide Search shows a parallel pattern from 1 to 5 retrievals per query, echoing the depth–breadth duality of the design. All other components are fixed at default to isolate this effect.

This non-monotonicity reveals a key property of stateful retrieval: more experience is not strictly better, since beyond a moderate budget, redundant guidance crowds the context window and dilutes the signal aligned with the current state. The Deep-and-Wide design therefore works because it balances coverage with parsimony, leveraging multiple semantic viewpoints without overwhelming the agent's instruction-following capacity.



(a) Deep Search

(b) Wide Search

Figure 2: Scaling behavior of Deep-and-Wide search on V* Bench and HR-Bench with Qwen3-VL-32B-Instruct, varying (a) Deep Search iterations L (with K=3) and (b) Wide Search retrievals K (with L=3).

Table 3: Cross-dataset experience transfer. For each target benchmark, we build a unified experience bank by pooling the exploration sets of the other three benchmarks (held out from the evaluation), simulating deployment on a previously unseen benchmark with no in-domain experience accumulation. We report the accuracy (%) for all tasks. Best results are highlighted in **bold**.

Method	Exp. Level	V* Bench			MME-RealWorld-Lite			Zoom-Bench	HR-Bench	Avg.
		Direct Attr.	Rel. Pos.	Micro Avg.	Percep.	Reas.	Micro Avg.			
Qwen3-VL-32B-Instruct										
Vanilla CoT (Kojima et al., 2022)	N/A	27.59	57.89	39.58	28.55	28.53	28.54	31.83	51.00	37.74
w/ ReAct (Yao et al., 2022)	N/A	62.07	**84.21**	70.83	44.44	36.80	41.46	44.69	66.00	55.75
w/ Reflexion (Shinn et al., 2023)	Trajectory	74.13	76.32	75.00	**46.15**	34.13	41.46	48.87	64.00	57.33
w/ Expel (Zhao et al., 2024)	Trajectory	70.69	78.95	73.96	42.39	**40.27**	**41.56**	46.30	66.00	56.96
w/ MuSEAgent (Cross-Dataset)	State	**77.59**	78.95	**78.12**	42.56	36.00	40.00	**51.45**	**69.00**	**59.64**

4.4 Cross-Dataset Experience Transfer

To evaluate whether stateful experiences transfer across datasets without any in-domain experience accumulation, we conduct cross-dataset experience transfer experiments with Qwen3-VL-32B-Instruct as the base model.

Despite operating without any in-domain interaction while baselines retain their in-domain experience banks, a deliberately conservative comparison that disadvantages MuSEAgent, our method using only cross-dataset experiences achieves an average accuracy of 59.64% (Table 3), still outperforming Reflexion (57.33%) and Expel (56.96%), both of which rely on target-domain trajectory data. This advantage is especially evident on benchmarks requiring complex visual operations: on Zoom-Bench and HR-Bench, MuSEAgent reaches 51.45% and 69.00%, exceeding the strongest in-domain trajectory-level baselines by 2.58% and 3.00% in absolute accuracy, respectively. The exception is MME-RealWorld-Lite (40.00%), where performance trails Expel (41.56%) and the zero-experience ReAct (41.46%), suggesting that targets with distinctive real-world visual characteristics still benefit from in-domain experience accumulation. Overall, the fact that cross-dataset stateful experiences surpass in-domain trajectory-level baselines indicates that decomposing trajectories into atomic state-action pairs captures reusable multimodal reasoning skills, rather than task-specific trajectory memorization. State-level retrieval transfers reusable tool-use tactics across perception-heavy benchmarks, whereas trajectory-level retrieval would drag along task-specific operator sequences; MME-RealWorld-Lite's heavier real-world priors limit such transfer. This pattern corroborates the state-level granularity adopted in Sec. 3.2: tactics at this granularity are generic enough to apply across tasks yet specific enough to remain actionable in new contexts.

Table 4: Ablations on Qwen3-VL-32B-Instruct over (a) experience source and (b) hindsight reasoning model $\mathcal{Q}_\phi$ from self-hindsight to stronger external teachers. We report the accuracy (%) for all tasks. Best results within each block are highlighted in **bold**. Sensitivity to the quality score threshold δ is reported in Appendix F.

Method	Exp. Level	Variant	V* Bench			HR-Bench
			Direct Attr.	Rel. Pos.	Micro Avg.	
Baselines						
Vanilla CoT (Kojima et al., 2022)	N/A	–	27.59	57.89	39.58	51.00
w/ ReAct (Yao et al., 2022)	N/A	–	62.07	84.21	70.83	66.00
w/ Reflexion (Shinn et al., 2023)	Trajectory	–	74.13	76.32	75.00	64.00
w/ Expel (Zhao et al., 2024)	Trajectory	–	70.69	78.95	73.96	66.00
(a) Experience Source						
w/ MuSEAgent	State	Correct only	62.07	**86.84**	71.88	72.00
w/ MuSEAgent	State	Incorrect only	75.86	76.32	76.04	76.00
w/ MuSEAgent	State	Correct + Incorrect	**86.21**	84.21	**85.42**	**78.00**
(b) Hindsight Reasoning Model $\mathcal{Q}_\phi$						
w/ MuSEAgent	State	Qwen3-VL-32B-Instruct (self)	75.86	81.58	78.12	75.00
w/ MuSEAgent	State	Qwen3-VL-235B-A22B-Instruct	74.14	81.58	77.08	76.00
w/ MuSEAgent	State	Qwen3.5-397B-A17B	77.59	**92.11**	83.33	73.00
w/ MuSEAgent	State	GPT-4o	**86.21**	84.21	**85.42**	**78.00**

4.5 Ablation Studies

Using Qwen3-VL-32B-Instruct on V* Bench and HR-Bench, we ablate two design choices and derive two findings:

Failure-derived experiences match or exceed success-derived ones. Block (a) of Table 4 compares trajectory sources. Success-only experiences yield 71.88% on V* Bench and 72.00% on HR-Bench; failure-only experiences raise these to 76.04% and 76.00%, indicating that correcting erroneous reasoning provides stronger guidance than reinforcing successful paths. Combining both reaches 85.42% and 78.00%, showing that a diverse bank lets the agent both learn optimal strategies and avoid known pitfalls.

Stateful experience gains hold under self-hindsight, ruling out a distillation confound. Block (b) varies the hindsight reasoning model $\mathcal{Q}_\phi$ to test whether gains depend on a strong external teacher. Under *self-hindsight*, where Qwen3-VL-32B-Instruct serves as both base and hindsight model, MuSEAgent already reaches 78.12% on V* Bench and 75.00% on HR-Bench, surpassing the strongest trajectory-level baselines (Reflexion at 75.00% and Expel at 66.00%) by +3.12 and +9.00 points. Scaling $\mathcal{Q}_\phi$ to Qwen3.5-397B-A17B raises V* Bench to 83.33%, and GPT-4o further reaches 85.42% and 78.00%. A stronger teacher therefore acts as a complementary upper bound rather than a precondition: the gains of stateful experience retrieval are not an artifact of distilling a proprietary teacher. Practically, this means MuSEAgent can be deployed without relying on any external API: under self-hindsight, the entire offline experience-bank construction reuses the same in-house inference stack as the base agent (See Appendix E for the efficiency details).

We also provide ablation analysis on sensitivity to the quality score threshold δ in Appendix F.

5 Conclusion

We present MuSEAgent, a multimodal reasoning agent that mitigates the noise and rigid context of trajectory-level retrieval by decomposing episodes into atomic state-action pairs. Hindsight reasoning yields a granular stateful experience bank, which the agent queries through a Deep-and-Wide search. Across diverse benchmarks and base model architectures, stateful guidance consistently outperforms trajectory-level methods. The gains hold even when the agent itself acts as the hindsight reasoner, and cross-dataset transfer further shows that state-level experiences capture reusable reasoning and tool-use skills, generalizing to held-out benchmarks without in-domain experience accumulation. Future work will explore online experience construction, letting agents continuously update their stateful memory and self-improve during interaction with open-ended environments.

Limitations

While MuSEAgent achieves consistent gains across multimodal reasoning benchmarks, the current experience bank is constructed once offline from a fixed exploration set and is not updated during deployment. As a result, novel failure modes encountered after deployment are not automatically incorporated into the bank, which motivates the online experience construction direction discussed in our conclusion. The current evaluation also focuses on tool-using reasoning over established perception and reasoning datasets. Extending the framework to dynamic interactive environments such as VisualWebArena or OSWorld is a natural next step, requiring the experience abstraction to generalize to action effects on persistent external state. Likewise, extending to broader modalities such as video, audio, or 3D scenes would enrich the compositional state representation with temporally extended or non-visual signals.

References

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, and 45 others. 2025. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*.

Ma Chang, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. 2024. Agentboard: An analytical evaluation board of multi-turn llm agents. *Advances in neural information processing systems*, 37:74325–74362.

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

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. 2024. Webvoyager: Building an end-to-end web agent with large multimodal models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6864–6890.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Mingxin Li, Yanzhao Zhang, Dingkun Long, Keqin Chen, Sibo Song, Shuai Bai, Zhibo Yang, Pengjun Xie, An Yang, Dayiheng Liu, and 1 others. 2026. Qwen3-vl-embedding and qwen3-vl-reranker: A unified framework for state-of-the-art multimodal retrieval and ranking. *arXiv preprint arXiv:2601.04720*.

Heng Lin and Zhongwen Xu. 2025. Understanding tool-integrated reasoning. *arXiv preprint arXiv:2508.19201*.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173.

Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. 2023. Memgpt: towards llms as operating systems.

Qwen Team. 2026. Qwen3.5: Towards native multimodal agents.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.

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

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

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, and 1 others. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345.

Wenbin Wang, Liang Ding, Minyan Zeng, Xiabin Zhou, Li Shen, Yong Luo, Wei Yu, and Dacheng Tao. 2025. Divide, conquer and combine: A training-free framework for high-resolution image perception in multimodal large language models. In *Proceedings*

of the AAAI Conference on Artificial Intelligence, volume 39, pages 7907–7915.

Lai Wei, Liangbo He, Jun Lan, Lingzhong Dong, Yutong Cai, Siyuan Li, Huijia Zhu, Weiqiang Wang, Linghe Kong, Yue Wang, Zhuosheng Zhang, and Weiran Huang. 2026. Zooming without zooming: Region-to-image distillation for fine-grained multimodal perception. *arXiv preprint arXiv:2602.11858*.

Penghao Wu and Saining Xie. 2024. V?: Guided visual search as a core mechanism in multimodal llms. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13084–13094.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, and 1 others. 2025. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, and 1 others. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094.

Zhengyuan Yang, Linjie Li, Jianfeng Wang, Kevin Lin, Ehsan Azarnasab, Faisal Ahmed, Zicheng Liu, Ce Liu, Michael Zeng, and Lijuan Wang. 2023. Mm-react: Prompting chatgpt for multimodal reasoning and action. *arXiv preprint arXiv:2303.11381*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Weiran Yao, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Yihao Feng, Le Xue, Rithesh Murthy, Zeyuan Chen, Jianguo Zhang, Devansh Arpit, and 1 others. 2023. Retroformer: Retrospective large language agents with policy gradient optimization. *arXiv preprint arXiv:2308.02151*.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2025. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pages 1–20.

Yi-Fan Zhang, Huanyu Zhang, Haochen Tian, Chaoyou Fu, Shuangqing Zhang, Junfei Wu, Feng Li, Kun Wang, Qingsong Wen, Zhang Zhang, and 1 others. 2024. Mme-realworld: Could your multimodal llm challenge high-resolution real-world scenarios that are difficult for humans? *arXiv preprint arXiv:2408.13257*.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19632–19642.

Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, and 1 others. 2025. Memento: Fine-tuning llm agents without fine-tuning llms. *arXiv preprint arXiv:2508.16153*.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, and 1 others. 2023. Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *arXiv preprint arXiv:2305.17144*.

A Algorithms

We provide the full procedures of the two phases of MuSEAgent and clarify how each algorithmic step instantiates the design choices motivated in Sec. 3. The two phases are deliberately decoupled: Algorithm 1 runs once offline over an exploration set of trajectories to materialize a reusable, indexable experience bank, while Algorithm 2 runs once per reasoning step at inference time and only touches the bank through similarity lookups. This decoupling is what allows hindsight scoring (which is comparatively expensive) to be amortized across all downstream queries, and makes cross-dataset transfer (Sec. 4.4) a drop-in operation: only the bank pointer changes.

Offline experience abstraction (Algorithm 1). The procedure ingests historical trajectories $\mathcal{T}$ and emits a quality-filtered, multi-viewpoint-indexed experience bank $\mathcal{B}$. We decompose every τ into atomic transitions (s_t, a_t, s_{t+1}), so that the basic unit of memory is a localized decision context rather than a full episode. No trajectory-level pre-filtering is applied: both successful and failed trajectories enter the pipeline, since failed ones often contain locally optimal sub-decisions and successful ones occasionally include detours. The hindsight reasoner $\mathcal{Q}_\phi$ resolves this at the transition level, producing a scalar quality q_t and a natural-language guidance g_t conditioned on the realized outcome o. Transitions with $q_t < \delta$ are discarded; transitions from failed trajectories are retained whenever $q_t \geq \delta$, since their guidance encodes *what to avoid* rather than *what to imitate*. Each surviving transition is wrapped into $e_t = (s_t, a_t, g_t)$ and indexed under all M viewpoints, because the inference-time viewpoint is state-adaptive and unknown offline; a missing index would force a re-embedding pass at deployment. The dominant offline cost is one call to $\mathcal{Q}_\phi$ per atomic transition plus $M \cdot |\mathcal{B}|$ embedding evaluations, both fully parallelizable and incurred once per exploration set.

Online Deep-and-Wide search (Algorithm 2). At inference step t, the agent consults $\mathcal{B}$ through up to L refinement rounds. In round j, the policy $\pi(\cdot \mid s_t)$ samples a viewpoint p_{i_j} emphasizing a different facet of s_t (e.g., perceptual intent under (u, v_t), tool-execution history under (d, H_t), or full task context (u, v_t, d, H_t)) and constructs $\mathbf{q}_t^{(i_j)}$. Viewpoint selection is performed by the base model via prompting, which keeps the framework compatible with any instruction-following base model and requires no additional training. Wide Search then retrieves the Top-K experiences from the matching viewpoint index by cosine similarity. The retrieved set $\mathcal{E}^{(j)}$ is unioned (with deduplication) into $\mathcal{E}_{\text{deep-wide}}$; union rather than replacement lets later rounds compensate for blind spots of earlier viewpoints. After L rounds, the retrieved guidances are concatenated into the working context and a_t is sampled jointly on s_t and the aggregated guidance. When $\mathcal{E}_{\text{deep-wide}}$ is empty, the agent falls back to its ReAct-style policy on s_t alone, so retrieval failure never blocks decision-making. With L and K small constants and embeddings pre-computed, the per-step cost is $\mathcal{O}(L \cdot |\mathcal{B}|)$ in the worst case, keeping deployment overhead bounded (Appendix E). Together, the two algorithms cleanly separate expensive hindsight-based bank construction from lightweight online retrieval, so improvements to the experience bank, the embedding model, or the base agent compose without architectural changes, and each component can be upgraded independently.

Algorithm 1 Stateful Experience Abstraction via Hindsight Reasoning

Require: Historical trajectories $\mathcal{T} = \{\tau_1, \tau_2, \dots\}$, quality threshold δ, candidate semantic viewpoints $\mathcal{P} = \{p_1, \dots, p_M\}$.
Ensure: Updated Experience Bank $\mathcal{B}$.

```
# Filter high-quality transitions and index each experience under M semantic viewpoints.
Initialize experience bank B ← ∅
for each trajectory τ ∈ T do
    for each atomic transition (s_t, a_t, s_{t+1}) ∈ τ do
        Compute hindsight quality score and guidance summary: (q_t, g_t) ← Q_φ(s_t, a_t, s_{t+1})
        if q_t ≥ δ then
            Construct stateful experience e_t ← (s_t, a_t, g_t)
            Build multi-viewpoint embeddings: z_t^(i) ← f_θ(p_i(s_t)), ∀ p_i ∈ P
            Update experience bank B ← B ∪ {e_t}
        end if
    end for
end for
return B
```

Algorithm 2 Experience Exploitation via Deep-and-Wide Search

Require: Experience Bank $\mathcal{B}$, current state s_t, retrieval breadth K, max refinement rounds L, candidate semantic viewpoints $\mathcal{P} = \{p_1, \ldots, p_M\}$.

Ensure: Execution action a_t.

Initialize unified guidance set $\mathcal{E}_{\text{deep-wide}} \leftarrow \emptyset$

 # Deep Search: Iterative Semantic Viewpoint Refinement

for round $j = 1$ to L **do**

  Sample semantic viewpoint and formulate query embedding: $p_{i_j} \sim \pi(\cdot \mid s_t)$, $\mathbf{q}_t^{(i_j)} \leftarrow f_\theta\big(p_{i_j}(s_t)\big)$

  # Wide Search: Top-K Retrieval under Current Viewpoint

  Retrieve top-K experiences:

$$\mathcal{E}^{(j)} \leftarrow \text{Top-K}_{e_m \in \mathcal{B}}\Big(\text{sim}\Big(\mathbf{q}_t^{(i_j)}, \mathbf{z}_m^{(i_j)}\Big)\Big)$$

  Aggregate retrieved experiences: $\mathcal{E}_{\text{deep-wide}} \leftarrow \mathcal{E}_{\text{deep-wide}} \cup \mathcal{E}^{(j)}$

end for

 # Action Generation

Generate execution action: $a_t \sim \pi\big(\cdot \mid s_t, \mathcal{E}_{\text{deep-wide}}\big)$

return a_t

B Datasets and Metrics

We summarize the benchmarks and corresponding metrics used to evaluate the fine-grained perception and high-resolution reasoning capabilities of multimodal models below:

V* Bench (Wu and Xie, 2024). This Visual Question Answering (VQA) benchmark is designed to evaluate the detailed visual grounding capabilities of multimodal models on high-resolution images. It consists of 191 carefully curated samples divided into two fine-grained sub-tasks: attribute recognition and spatial relationship reasoning. All questions are presented in a multiple-choice format. The benchmark tests whether models can accurately perceive small, critical visual details that standard vision encoders typically overlook. We use the overall accuracy across these sub-tasks as the final score.

MME-RealWorld-Lite (Zhang et al., 2024). This is an accelerated evaluation subset of the large-scale MME-RealWorld benchmark, which tests models on high-resolution images across challenging, practical real-world scenarios. By efficiently subsampling up to 50 instances across 43 tasks, the Lite version maintains diverse scenario coverage with reduced computational cost. For our evaluation, we select the multiple-choice questions from this dataset, totaling 1,919 items. We use the average task accuracy as the final score.

Zoom-Bench (Wei et al., 2026). Consisting of 845 instances, this VQA benchmark is designed to assess fine-grained multimodal perception. It specifically focuses on challenging scenarios where the decisive visual evidence is extremely small and easily overshadowed by the global context, evaluating a model's ability to accurately capture minute details. In our evaluation, we select the multiple-choice subset, which contains 621 questions. We use the overall VQA accuracy as the final score.

HR-Bench (Wang et al., 2025). This benchmark is deliberately constructed to measure multimodal models' perception and reasoning capabilities on true high-resolution images. It focuses on fine-grained single-instance and cross-instance perception tasks, systematically evaluating how well models extract and reason over intricate visual details typically lost during standard image down-sampling. In our evaluation, we select 200 multiple-choice questions based on 8K-resolution images from HR-Bench, and use the overall accuracy on these high-resolution questions as the final score.

C Baseline Methods

The baseline methods we compare against in our experiments are summarized as follows:

Vanilla CoT (Kojima et al., 2022). Chain-of-Thought prompting directs LLMs/VLMs to generate a sequence of intermediate reasoning steps before producing a final answer. This method relies solely on the model's pre-trained parametric knowledge to decompose complex problems into sequential logical deductions. Operating without external tool invocations or environmental feedback, Vanilla CoT serves as a baseline to evaluate the intrinsic reasoning capacity and standalone problem-solving performance of language models.

ReAct (Yao et al., 2022). This framework integrates reasoning and acting within LLMs/VLMs.

Instead of relying solely on internal knowledge, ReAct enables models to generate reasoning traces and task-specific actions in an interleaved manner. This process allows the agent to formulate and adjust plans while interacting with external environments, such as knowledge bases or APIs, to retrieve real-time observations. By grounding its reasoning in external information, ReAct provides a mechanism to reduce factual errors and maintain interpretability during multi-step decision-making tasks.

Reflexion (Shinn et al., 2023). Reflexion equips agents with a mechanism for improving their behavior through verbal self-reflection. After completing a task attempt, an evaluator analyzes the resulting trajectory and produces natural language reflections based on environmental feedback. These reflections are stored in an episodic memory buffer and provided as additional context in subsequent attempts, enabling the agent to revise its reasoning strategy and avoid repeating previous mistakes. In our implementation, reflective experiences are extracted only from failed trajectories.

Expel (Zhao et al., 2024). Expel is a method that enables agents to distill reusable knowledge from their interactions with the environment. By analyzing task trajectories, the agent extracts generalized rules or strategies that can guide future decision making. Unlike approaches that focus primarily on failure analysis, Expel leverages information from both successful and unsuccessful experiences to identify patterns that improve planning and reasoning. The extracted knowledge is stored in an experience pool and incorporated into future task executions. In our implementation, experiences are extracted from both successful and failed trajectories.

D Tool Descriptions

We equip our model with a diverse set of tools to interact with visual inputs, perform calculations, and access external information. These tools are categorized into four functional modules below.

D.1 Image Manipulation Tools

Zoom In. This tool scales a specific region of an image based on a provided 2D bounding box and a numerical zoom factor. It outputs the magnified image segment to facilitate the inspection of minute details.

```
class ZoomInTool(BasicTool):
    name = "zoom_in"
    description_en = "Zoom in on a
    region by cropping and upscaling.
    Use localize_objects first if bbox
    is unknown."

    parameters = {
        "type": "object",
        "properties": {
            "image": {
                "type": "string",
                "description": "Image ID
     (e.g., 'img_0')"
            },
            "bbox_2d": {
                "type": "array",
                "items": {"type": "
    number"},
                "minItems": 4,
                "maxItems": 4,
                "description": "Bounding
     box [left, top, right, bottom],
    values between 0 and 1"
            },
            "zoom_factor": {
                "type": "number",
                "description": "Zoom
    factor (must be > 1)",
                "exclusiveMinimum": 1.0
            }
        },
        "required": ["image", "bbox_2d",
     "zoom_factor"]
    }
    example = '{"image": "img_0", "
    bbox_2d": [0.25, 0.25, 0.75, 0.75],
    "zoom_factor": 2.0}'

    def call(self, params: Union[str,
    Dict]) -> Dict:
        params_dict = self.parse_params(
    params)

        image_path = params_dict["image"
    ]
        bbox_2d = params_dict["bbox_2d"]
        zoom_factor = params_dict["
    zoom_factor"]

        if not all(0 <= x <= 1.0 for x
    in bbox_2d):
            return {"error": "bbox_2d
    values must be between 0 and 1"}
        if bbox_2d[0] >= bbox_2d[2] or
    bbox_2d[1] >= bbox_2d[3]:
            return {"error": "invalid
    bbox_2d: left >= right or top >=
    bottom"}
        if zoom_factor <= 1:
            return {"error": "
    zoom_factor must be > 1"}

        try:
            image = image_processing(
    image_path)
        except Exception as e:
            return {"error": f"failed to
     load image: {e}"}
```

```
        W, H = image.size
        crop_box = (int(bbox_2d[0] * W), int(bbox_2d[1] * H), int(bbox_2d[2] * W), int(bbox_2d[3] * H))
        cropped = image.crop(crop_box)

        new_size = (int(cropped.width * zoom_factor), int(cropped.height * zoom_factor))
        zoomed = cropped.resize(new_size, Image.LANCZOS)

        return {"output_image": zoomed}

    def generate_description(self, properties, observation):
        img = properties.get("image", "image")
        bbox_2d = properties.get("bbox_2d", [])
        factor = properties.get("factor", 2)
        return f"Zoomed in {img} at bbox_2d {bbox_2d} by {factor}x"
```

Crop. This tool extracts a sub-region from the original image using specified 2D bounding box coordinates. It removes peripheral visual context to isolate and return the target area as a new image.

```
class CropTool(BasicTool):
    name = "crop"
    description_en = "Crop a region from an image. Use localize_objects first if bbox is unknown."

    parameters = {
        "type": "object",
        "properties": {
            "image": {
                "type": "string",
                "description": "Image ID (e.g., 'img_0')"
            },
            "bbox_2d": {
                "type": "array",
                "items": {"type": "number"},
                "minItems": 4,
                "maxItems": 4,
                "description": "Bounding box as [left, top, right, bottom], values between 0 and 1"
            }
        },
        "required": ["image", "bbox_2d"]
    }
    example = '{"image": "img_0", "bbox_2d": [0.1, 0.2, 0.5, 0.6]}'

    def call(self, params: Union[str, Dict]) -> Dict:
        params_dict = self.parse_params(params)

        image_path = params_dict["image"]
        bbox_2d = params_dict["bbox_2d"]
        if not all(0 <= x <= 1.0 for x in bbox_2d):
            return {"error": "bbox_2d values must be between 0 and 1"}
        if bbox_2d[0] >= bbox_2d[2] or bbox_2d[1] >= bbox_2d[3]:
            return {"error": "invalid bbox_2d: left >= right or top >= bottom"}

        try:
            image = image_processing(image_path)
        except Exception as e:
            return {"error": f"failed to load image: {e}"}

        W, H = image.size
        crop_box = (int(bbox_2d[0] * W), int(bbox_2d[1] * H), int(bbox_2d[2] * W), int(bbox_2d[3] * H))
        cropped = image.crop(crop_box)

        return {"output_image": cropped}

    def generate_description(self, properties, observation):
        img = properties.get("image", "image")
        bbox_2d = properties.get("bbox_2d", [])
        return f"Cropped {img} at bbox_2d {bbox_2d}"
```

Visualize Regions. This tool overlays annotated bounding boxes and their corresponding text labels directly onto an input image. It generates a new visual output that explicitly highlights the specified regions of interest for structural grounding.

```
class VisualizeRegionsOnImageTool(BasicTool):
    name = "visualize_regions"
    description_en = "Draw bounding boxes and labels on an image. Each region is defined by normalized coordinates [x1, y1, x2, y2] where values are between 0 and 1. Use localize_objects first if bbox is unknown."

    parameters = {
        "type": "object",
        "properties": {
            "image": {
                "type": "string",
                "description": "Image ID (e.g., 'img_0')"
            },
            "regions": {
                "type": "array",
                "description": "List of regions to label, each with 'bbox_2d' and optional 'label'",
                "items": {
                    "type": "object",
                    "properties": {
```

```
                        "bbox_2d": {
                            "type": "
array",
                            "description
": "Normalized bounding box [x1, y1,
 x2, y2] where values are between 0
and 1",
                            "items": {"
type": "number"},
                            "minItems":
4,
                            "maxItems":
4
                        },
                        "label": {
                            "type": "
string",
                            "description
": "Optional label text for the
region"
                        }
                    },
                    "required": ["
bbox_2d"]
                }
            },
            "color": {
                "type": "string",
                "description": "Color of
 the bounding boxes (default: '
yellow')"
            },
            "width": {
                "type": "integer",
                "description": "Width of
 the bounding box lines (default: 4)
"
            }
        },
        "required": ["image", "regions"]
}
example = '{"image": "img_0", "
regions": [{"bbox_2d": [0.1, 0.1,
0.5, 0.5], "label": "Object A"}, {"
bbox_2d": [0.6, 0.6, 0.9, 0.9], "
label": "Object B"}]}'

def call(self, params: Union[str,
Dict]) -> str:
    if not PIL_AVAILABLE:
        return {"error": "PIL (
Pillow) is not installed. Please
install it with: pip install Pillow"
}

    params_dict = self.parse_params(
params)

    image_path = params_dict["image"
]
    regions = params_dict["regions"]
    color = params_dict.get("color",
 "yellow")
    width = params_dict.get("width",
 4)

    try:
        image = image_processing(
image_path)
        W, H = image.size
        img_labeled = image.copy()
        draw = ImageDraw.Draw(
img_labeled)

        try:
            font = ImageFont.
truetype('/usr/share/fonts/truetype/
dejavu/DejaVuSansMono-Bold.ttf', 16)
        except (OSError, IOError):
            try:
                font = ImageFont.
truetype('/System/Library/Fonts/
Helvetica.ttc', 16)
            except (OSError, IOError
):
                font = ImageFont.
load_default()

        text_color = 'black'

        for obj in regions:
            bbox_2d = obj['bbox_2d']
            bbox_pixels = (bbox_2d
[0] * W, bbox_2d[1] * H, bbox_2d[2]
* W, bbox_2d[3] * H)

            draw.rectangle(
bbox_pixels, outline=color, width=
width)

            x1, y1, x2, y2 =
bbox_pixels
            label = obj.get('label',
 '')

            if label:
                try:
                    bbox_text = draw
.textbbox((0, 0), label, font=font)
                    w = bbox_text[2]
 - bbox_text[0]
                    h = bbox_text[3]
 - bbox_text[1]
                except
AttributeError:
                    w, h = font.
getsize(label)

                if x1 + w > W or y2
+ h > H:
                    draw.rectangle((
x1, y2 - h, x1 + w, y2), fill=color)
                    draw.text((x1,
y2 - h), label, fill=text_color,
font=font)
                else:
                    draw.rectangle((
x1, y2, x1 + w, y2 + h), fill=color)
                    draw.text((x1,
y2), label, fill=text_color, font=
font)

        return {"output_image":
img_labeled}

    except FileNotFoundError:
        return {"error": f"Image
file not found at {image_path}"}
    except Exception as e:
```

```
            return {"error": str(e)}

    def generate_description(self, properties, observation):
        img = properties.get("image", "image")
        regions = properties.get("regions", [])
        num_regions = len(regions) if isinstance(regions, list) else 0
        return f"Visualized {num_regions} regions on {img}"
```

D.2 Visual Perception Tools

OCR. Optical Character Recognition scans an input image to detect and extract visible text elements. It returns the recognized textual content to assist in reading and parsing tasks.

```
OCR_PROMPT = "OCR:"
MAX_PIXELS_OCR = 1280 * 28 * 28

class OCRTool(ModelBasedTool):
    name = "ocr"
    model_id = "ocr"

    description_en = "Extract text from an image. Returns empty if no text found."

    parameters = {
        "type": "object",
        "properties": {
            "image": {
                "type": "string",
                "description": "Image ID (e.g., 'img_0')"
            }
        },
        "required": ["image"]
    }
    example = '{"image": "img_0"}'

    def load_model(self, device: str) -> None:
        self.model = AutoModelForImageTextToText.from_pretrained(
            PADDLE_OCR_VL_MODEL_PATH, torch_dtype=torch.bfloat16
        ).to(device).eval()
        self.processor = AutoProcessor.from_pretrained(PADDLE_OCR_VL_MODEL_PATH)
        self.device = device
        self.is_loaded = True

    def _call_impl(self, params: Union[str, Dict]) -> Dict:
        params_dict = self.parse_params(params)
        image_path = params_dict["image"]
        tmp_file_to_remove = None

        try:
            full_path = image_path if os.path.exists(image_path) else get_full_path_data(image_path)
            if full_path and os.path.exists(full_path):
                image_file = full_path
            else:
                image = image_processing(image_path)
                if isinstance(image, Image.Image):
                    tmp = tempfile.NamedTemporaryFile(suffix=".png", delete=False)
                    tmp_file_to_remove = tmp.name
                    try:
                        image.save(tmp.name)
                        image_file = tmp.name
                    finally:
                        tmp.close()
                elif isinstance(image, str):
                    resolved = image if os.path.exists(image) else get_full_path_data(image)
                    if resolved is None:
                        raise FileNotFoundError(f"Image not found: {image_path}")
                    image_file = resolved
                else:
                    raise ValueError(f"Unexpected image type: {type(image)}")

            image = Image.open(image_file).convert("RGB")

            messages = [
                {
                    "role": "user",
                    "content": [
                        {"type": "image", "image": image},
                        {"type": "text", "text": OCR_PROMPT},
                    ]
                }
            ]
            inputs = self.processor.apply_chat_template(
                messages,
                add_generation_prompt=True,
                tokenize=True,
                return_dict=True,
                return_tensors="pt",
                images_kwargs={
                    "size": {
                        "shortest_edge": self.processor.image_processor.min_pixels,
                        "longest_edge": MAX_PIXELS_OCR,
                    }
                },
            )
```

```
            if "pixel_values" in inputs and not isinstance(inputs["pixel_values"], torch.Tensor):
                inputs["pixel_values"] = torch.from_numpy(inputs["pixel_values"]).to(self.device)
            if "image_grid_thw" in inputs and not isinstance(inputs["image_grid_thw"], torch.Tensor):
                inputs["image_grid_thw"] = torch.from_numpy(inputs["image_grid_thw"]).to(self.device)

            inputs = {
                k: v.to(self.device) if isinstance(v, torch.Tensor) else v
                for k, v in inputs.items()
            }

            outputs = self.model.generate(**inputs, max_new_tokens=512)
            extracted_text = self.processor.decode(
                outputs[0][inputs["input_ids"].shape[-1] : -1]
            ).strip()

            return {"extracted text": extracted_text}
        except FileNotFoundError as e:
            return {"error": f"Image not found: {str(e)}"}
        except Exception as e:
            return {"error": f"OCR error: {str(e)}\n{traceback.format_exc()}"}
        finally:
            if tmp_file_to_remove and os.path.exists(tmp_file_to_remove):
                try:
                    os.unlink(tmp_file_to_remove)
                except OSError:
                    pass
```

Localize Objects. This tool identifies the spatial locations of user-specified object categories within an image. It processes the semantic queries and outputs the 2D bounding boxes corresponding to the detected entities.

```
class LocalizeObjectsTool(ModelBasedTool):
    name = "localize_objects"
    model_id = "sam3"

    description_en = "Localize objects and return their bounding boxes. Use this to get bbox for other region-based tools. Note: Cannot detect text labels or annotation markers (e.g., 'A', 'B', 'point 1') drawn on images."

    parameters = {
        "type": "object",
        "properties": {
            "image": {"type": "string", "description": "Image ID (e.g., 'img_0')"},
            "objects": {"type": "array", "items": {"type": "string"}, "description": "A list of object names to localize. e.g. ['dog', 'cat', 'person']."}
        },
        "required": ["image", "objects"]
    }
    example = '{"image": "img_0", "objects": ["dog", "cat"]}'

    def load_model(self, device: str) -> None:
        self.processor = Sam3Processor.from_pretrained(SAM3_MODEL_PATH)
        self.model = Sam3Model.from_pretrained(SAM3_MODEL_PATH).to(device)
        self.device = device
        self.is_loaded = True

    def _call_impl(self, params: Union[str, Dict]) -> str:
        params_dict = self.parse_params(params)
        image_path = params_dict["image"]
        objects = params_dict["objects"]

        if not isinstance(objects, list) or len(objects) == 0:
            return {
                "error": "objects must be a non-empty list of strings"
            }

        try:
            image = image_processing(image_path)
            W, H = image.size

            regions = []
            obj_cnt = {}

            for obj_name in objects:
                inputs = self.processor(images=image, text=obj_name, return_tensors="pt").to(self.device)

                with torch.no_grad():
                    outputs = self.model(**inputs)

                results = self.processor.post_process_instance_segmentation(
                    outputs,
                    threshold=0.5,
                    mask_threshold=0.5,
                    target_sizes=inputs.get("original_sizes").tolist()
                )[0]

                boxes = results["boxes"]
                scores = results["scores"]
```

```
                for box, score in zip(
boxes, scores):
                    score_val = score.
item()
                    if score_val < 0.50:
                        continue

                    box_list = box.
tolist()
                    bbox = [
                        round(float(
box_list[0]) / W, 4),
                        round(float(
box_list[1]) / H, 4),
                        round(float(
box_list[2]) / W, 4),
                        round(float(
box_list[3]) / H, 4)
                    ]

                    obj_cnt[obj_name] =
obj_cnt.get(obj_name, 0) + 1
                    label_out = f"{
obj_name}-{obj_cnt[obj_name]}" if
obj_cnt[obj_name] > 1 else obj_name
                    regions.append({
                        "label":
label_out,
                        "bbox_2d": bbox,
                        "score": round(
score_val, 4)
                    })

            visualize_tool =
VisualizeRegionsOnImageTool()
            visualize_params = {
                "image": image_path,
                "regions": [{"bbox_2d":
r["bbox_2d"], "label": r["label"]}
for r in regions]
            }
            output_image_result =
visualize_tool.call(visualize_params
)
            output_image =
output_image_result.get("
output_image")

            return {
                "output_image":
output_image,
                "regions": regions
            }

        except FileNotFoundError as e:
            return {"error": f"Image
file not found: {str(e)}"}
        except Exception as e:
            return {"error": f"Error
localizing objects: {str(e)}"}

    def generate_description(self,
properties, observation):
        img = properties.get("image", "
image")
        objects = properties.get("
objects", [])
        if isinstance(objects, list):
            objects_str = ", ".join(
objects) if objects else "objects"
        else:
            objects_str = str(objects)
        return f"Localized {objects_str}
 in {img}"
```

Estimate Region Depth. This tool calculates the spatial depth values within a designated bounding box. It aggregates the pixel-level depth maps and outputs statistical results, such as the mean depth of the specified area.

```
class EstimateRegionDepthTool(
    ModelBasedTool):
    name = "estimate_region_depth"
    model_id = "depth_anything"

    description_en = "Estimate depth of
    a region. Smaller value = closer.
    Use localize_objects first if bbox
    is unknown."

    parameters = {
        "type": "object",
        "properties": {
            "image": {
                "type": "string",
                "description": "Image ID
 (e.g., 'img_0')"
            },
            "bbox_2d": {
                "type": "array",
                "items": {"type": "
number"},
                "minItems": 4,
                "maxItems": 4,
                "description": "Bounding
 box [x1, y1, x2, y2] in normalized
coordinates (0-1)"
            },
            "mode": {
                "type": "string",
                "enum": ["mean", "min",
"max"],
                "description": "Mode to
compute depth: mean (average), min (
closest), or max (farthest)",
                "default": "mean"
            }
        },
        "required": ["image", "bbox_2d"]
    }
    example = '{"image": "img_0", "
bbox_2d": [0.1, 0.2, 0.5, 0.6], "
mode": "mean"}'

    def load_model(self, device: str) ->
 None:
        model_id = DEPTH_ANYTHING_PATH
        self.image_processor =
AutoImageProcessor.from_pretrained(
model_id)
        self.model =
AutoModelForDepthEstimation.
from_pretrained(model_id)
        self.model.to(device)
        self.model.eval()
        self.device = device
        self.is_loaded = True
```

```
    def _call_impl(self, params: Union[
str, Dict]) -> str:
        params_dict = self.parse_params(
params)
        image_path = params_dict["image"
]
        bbox_2d = params_dict["bbox_2d"]
        mode = params_dict.get("mode", "
mean")

        if not all(0 <= x <= 1.0 for x
in bbox_2d):
            return {"error": "bbox_2d
values must be between 0 and 1"}
        if bbox_2d[0] >= bbox_2d[2] or
bbox_2d[1] >= bbox_2d[3]:
            return {"error": "invalid
bbox_2d: left >= right or top >=
bottom"}

        try:
            image = image_processing(
image_path)
            W, H = image.size

            x1 = int(bbox_2d[0] * W)
            y1 = int(bbox_2d[1] * H)
            x2 = int(bbox_2d[2] * W)
            y2 = int(bbox_2d[3] * H)

            inputs = self.
image_processor(images=image,
return_tensors="pt")
            inputs = {k: v.to(self.
device) for k, v in inputs.items()}

            with torch.no_grad():
                outputs = self.model(**
inputs)

            processed_outputs = self.
image_processor.
post_process_depth_estimation(
                outputs,
                target_sizes=[(H, W)],
            )
            depth_map =
processed_outputs[0]["
predicted_depth"].squeeze().detach()
.cpu().numpy()
            depth_map = depth_map.max()
- depth_map

            region_depth = depth_map[y1:
y2, x1:x2]

            if mode == "mean":
                depth_value = float(np.
mean(region_depth))
            elif mode == "min":
                depth_value = float(np.
min(region_depth))
            elif mode == "max":
                depth_value = float(np.
max(region_depth))
            else:
                depth_value = float(np.
mean(region_depth))

            return {"estimated depth":
round(depth_value, 4)}

        except FileNotFoundError as e:
            return {"error": f"Image
file not found: {str(e)}"}
        except Exception as e:
            return {"error": f"Error
estimating depth: {str(e)}"}
```

Estimate Object Depth. This tool predicts the depth value of a specific semantic object within the visual scene. It combines object recognition with monocular depth estimation to locate the target and determine its distance.

```
class EstimateObjectDepthTool(BasicTool)
    :
    name = "estimate_object_depth"
    description_en = "Estimate depth of
    an object by text description.
    Smaller value = closer."

    parameters = {
        "type": "object",
        "properties": {
            "image": {
                "type": "string",
                "description": "Image ID
     (e.g., 'img_0')"
            },
            "object": {
                "type": "string",
                "description": "A short
    description of the object to get the
     depth from"
            },
            "mode": {
                "type": "string",
                "enum": ["mean", "min",
    "max"],
                "description": "Mode to
    compute depth: mean (average), min (
    closest), or max (farthest)",
                "default": "mean"
            }
        },
        "required": ["image", "object"]
    }
    example = '{"image": "img_0", "
    object": "a black cat", "mode": "
    mean"}'

    def call(self, params: Union[str,
    Dict]) -> Dict:
        params_dict = self.parse_params(
    params)
        image_path = params_dict["image"
    ]
        object_desc = params_dict["
    object"]
        mode = params_dict.get("mode", "
    mean")

        try:
            localize_tool_class =
    TOOL_REGISTRY.get("localize_objects"
    )
```

```
        if localize_tool_class is
None:
            return {
                "error": "
localize_objects tool is not
available"
            }

        localize_tool =
localize_tool_class()
        localize_params = {
            "image": image_path,
            "objects": [object_desc]
        }
        localize_result =
localize_tool.call(localize_params)

        if isinstance(
localize_result, str):
            localize_data = json.
loads(localize_result)
        else:
            localize_data =
localize_result

        if "error" in localize_data
or len(localize_data.get("regions",
[])) == 0:
            return {
                "error": "Object not
 found"
            }

        regions = localize_data["
regions"]
        best_match_idx = np.argmax([
region.get("score", 0) for region in
 regions])
        bbox_2d = regions[
best_match_idx]["bbox_2d"]

        estimate_depth_tool_class =
TOOL_REGISTRY.get("
estimate_region_depth")
        if estimate_depth_tool_class
 is None:
            return {"error": "
estimate_region_depth tool is not
available"}

        estimate_depth_tool =
estimate_depth_tool_class()
        depth_params = {
            "image": image_path,
            "bbox_2d": bbox_2d,
            "mode": mode
        }
        return estimate_depth_tool.
call(depth_params)

    except Exception as e:
        return {"error": f"Error
estimating object depth: {str(e)}"}
```

D.3 Semantic Similarity Tools

Image-to-Image Similarity. This tool computes the visual embedding distance between a source image and a set of reference images. It outputs similarity scores to evaluate the visual resemblance among different image inputs.

```
class GetImageToImagesSimilarityTool(
    ModelBasedTool):
    name = "get_image2images_similarity"
    model_id = "clip"

    description_en = "Compute similarity
     between one image and multiple
    images."

    parameters = {
        "type": "object",
        "properties": {
            "image": {
                "type": "string",
                "description": "Image ID
 (e.g., 'img_0')"
            },
            "other_images": {
                "type": "array",
                "items": {"type": "
string"},
                "description": "The
other images to compare to the
reference image"
            }
        },
        "required": ["image", "
other_images"]
    }
    example = '{"image": "img_0", "
other_images": ["img_1", "img_2"]}'

    def load_model(self, device: str) ->
 None:
        self.model, _, self.preprocess =
 open_clip.
create_model_and_transforms(
CLIP_VERSION, pretrained=
CLIP_PRETRAINED)
        self.model.eval()
        self.model = self.model.to(
device)
        self.tokenizer = open_clip.
get_tokenizer(CLIP_VERSION)
        self.device = device
        self.is_loaded = True

    def _call_impl(self, params: Union[
str, Dict]) -> str:
        params_dict = self.parse_params(
params)
        image_path = params_dict["image"
]
        other_images = params_dict["
other_images"]

        if not isinstance(other_images,
list) or len(other_images) == 0:
            return {
                "error": "other_images
must be a non-empty list of image
paths"
            }

        try:
            reference_image =
image_processing(image_path)
```

```
            reference_image_tensor =
self.preprocess(reference_image).
unsqueeze(0).to(self.device)

            other_images_processed = []
            for img_path in other_images
:
                img = image_processing(
img_path)
                other_images_processed.
append(img)

            other_images_tensor = torch.
stack([
                self.preprocess(img).to(
self.device) for img in
other_images_processed
            ])

            with torch.no_grad(), torch.
cuda.amp.autocast():
                reference_features =
self.model.encode_image(
reference_image_tensor)
                other_features = self.
model.encode_image(
other_images_tensor)

                reference_features /=
reference_features.norm(dim=-1,
keepdim=True)
                other_features /=
other_features.norm(dim=-1, keepdim=
True)

                similarity_scores =
reference_features @ other_features.
T

            sim_scores = [round(score.
item(), 2) for score in
similarity_scores[0]]

            best_image_index = torch.
argmax(similarity_scores, dim=1).
item()

            return {
                "similarity scores":
sim_scores,
                "best match":
other_images[best_image_index]
            }

        except FileNotFoundError as e:
            return {"error": f"Image
file not found: {str(e)}"}
        except Exception as e:
            return {"error": f"Error
computing image similarity: {str(e)}
"}
```

Image-to-Text Similarity. This tool evaluates the semantic alignment between a given image and multiple textual descriptions. It returns scores indicating how accurately each text matches the visual content.

```
class GetImageToTextsSimilarityTool(
    ModelBasedTool):
    name = "get_image2texts_similarity"
    model_id = "clip"

    description_en = "Compute similarity
     between one image and multiple
    texts (image  texts)."

    parameters = {
        "type": "object",
        "properties": {
            "image": {
                "type": "string",
                "description": "Image ID
     (e.g., 'img_0')"
            },
            "texts": {
                "type": "array",
                "items": {"type": "
    string"},
                "description": "A list
    of texts to compare to the reference
     image"
            }
        },
        "required": ["image", "texts"]
    }
    example = '{"image": "img_0", "texts
    ": ["a cat", "a dog"]}'

    def load_model(self, device: str) ->
     None:
        self.model, _, self.preprocess =
     open_clip.
    create_model_and_transforms(
    CLIP_VERSION, pretrained=
    CLIP_PRETRAINED)
        self.model.eval()
        self.model = self.model.to(
    device)
        self.tokenizer = open_clip.
    get_tokenizer(CLIP_VERSION)
        self.device = device
        self.is_loaded = True

    def _call_impl(self, params: Union[
    str, Dict]) -> str:
        params_dict = self.parse_params(
    params)
        image_path = params_dict["image"
    ]
        texts = params_dict["texts"]

        if not isinstance(texts, list)
    or len(texts) == 0:
            return {
                "error": "texts must be
    a non-empty list of strings"
            }

        try:
            reference_image =
    image_processing(image_path)
            reference_image_tensor =
    self.preprocess(reference_image).
    unsqueeze(0).to(self.device)

            text_tokens = self.tokenizer
    (texts).to(self.device)
```

```
            with torch.no_grad(), torch.
cuda.amp.autocast():
                image_features = self.
model.encode_image(
reference_image_tensor)
                text_features = self.
model.encode_text(text_tokens)

                image_features /=
image_features.norm(dim=-1, keepdim=
True)
                text_features /=
text_features.norm(dim=-1, keepdim=
True)

                similarity_scores =
image_features @ text_features.T

            sim_scores = [round(score.
item(), 2) for score in
similarity_scores[0]]

            best_text_index = torch.
argmax(similarity_scores, dim=1).
item()

            return {
                "similarity scores":
sim_scores,
                "best match": texts[
best_text_index]
            }

        except FileNotFoundError as e:
            return {"error": f"Image
file not found: {str(e)}"}
        except Exception as e:
            return {"error": f"Error
computing image-to-text similarity:
{str(e)}"}
```

Text-to-Image Similarity. This tool measures how well a set of images corresponds to a specific text query. It calculates alignment scores for each image, functioning as a targeted text-based image retrieval metric.

```
class GetTextToImagesSimilarityTool(
    ModelBasedTool):
    name = "get_text2images_similarity"
    model_id = "clip"

    description_en = "Compute similarity
     between one text and multiple
    images (text  images)."

    parameters = {
        "type": "object",
        "properties": {
            "text": {
                "type": "string",
                "description": "The
reference text"
            },
            "images": {
                "type": "array",
                "items": {"type": "
string"},
                "description": "A list
of images to compare to the
reference text"
            }
        },
        "required": ["text", "images"]
    }
    example = '{"text": "a black and
white cat", "images": ["img_0", "
img_1"]}'

    def load_model(self, device: str) ->
     None:
        self.model, _, self.preprocess =
     open_clip.
create_model_and_transforms(
CLIP_VERSION, pretrained=
CLIP_PRETRAINED)
        self.model.eval()
        self.model = self.model.to(
device)
        self.tokenizer = open_clip.
get_tokenizer(CLIP_VERSION)
        self.device = device
        self.is_loaded = True

    def _call_impl(self, params: Union[
str, Dict]) -> str:
        params_dict = self.parse_params(
params)
        text = params_dict["text"]
        images = params_dict["images"]

        if not isinstance(images, list)
or len(images) == 0:
            return {
                "error": "images must be
 a non-empty list of image paths"
            }

        try:
            text_tokens = self.tokenizer
([text]).to(self.device)

            images_processed = []
            for img_path in images:
                img = image_processing(
img_path)
                images_processed.append(
img)

            images_tensor = torch.stack
([
                self.preprocess(img).to(
self.device) for img in
images_processed
            ])

            with torch.no_grad(), torch.
cuda.amp.autocast():
                text_features = self.
model.encode_text(text_tokens)
                image_features = self.
model.encode_image(images_tensor)

                text_features /=
text_features.norm(dim=-1, keepdim=
True)
                image_features /=
image_features.norm(dim=-1, keepdim=
```

```
True)

            similarity_scores =
text_features @ image_features.T

        sim_scores = [round(score.
item(), 2) for score in
similarity_scores[0]]

        best_image_index = torch.
argmax(similarity_scores, dim=1).
item()

        return {
            "similarity scores":
sim_scores,
            "best match": images[
best_image_index]
        }

    except FileNotFoundError as e:
        return {"error": f"Image
file not found: {str(e)}"}
    except Exception as e:
        return {"error": f"Error
computing text-to-image similarity:
{str(e)}"}
```

D.4 Reasoning and External Tools

Calculator. This tool evaluates standard arithmetic expressions provided as strings. It processes mathematical operators and numbers to return the exact numerical calculation result.

```
class CalculatorTool(BasicTool):
    name = "calculator"
    description_en = "Calculate
    mathematical expressions. Supports
    basic arithmetic (+, -, *, /),
    exponentiation (**), and parentheses
    ."

    parameters = {
        "type": "object",
        "properties": {
            "expression": {
                "type": "string",
                "description": "The
    mathematical expression to calculate
    "
            }
        },
        "required": ["expression"]
    }
    example = '{"expression": "123 * 456
     + 789"}'

    def call(self, params: Union[str,
    Dict]) -> Dict:
        params_dict = self.parse_params(
    params)
        expression = params_dict["
    expression"]

        if not self._is_safe_expression(
    expression):
            return {"error": "Expression
     contains invalid or unsafe
    characters"}

        try:
            result = eval(expression, {"
    __builtins__": {}}, {})
            return {"calculation result"
    : result}
        except ZeroDivisionError:
            return {"error": "Division
    by zero"}
        except SyntaxError:
            return {"error": "Invalid
    syntax in expression"}
        except Exception as e:
            return {"error": str(e)}

    def _is_safe_expression(self, expr:
    str) -> bool:
        safe_pattern = re.compile(r'^[\d
    \s\+\-\*\/\(\)\.\*\*]+$')
        return bool(safe_pattern.match(
    expr))
```

Solve Math Equation. This tool processes and solves algebraic equations or math-related word problems presented in natural language formats. It executes the mathematical logic and outputs the analytical or numerical solution.

```
class SolveMathEquationTool(BasicTool):
    name = "solve_math_equation"
    description_en = (
        "Solve mathematical equations
    and problems using WolframAlpha. "
        "Supports algebra, calculus, and
     symbolic/numeric math queries."
    )

    parameters = {
        "type": "object",
        "properties": {
            "query": {
                "type": "string",
                "description": "A
    mathematical question or equation to
     solve",
            }
        },
        "required": ["query"],
    }
    example = '{"query": "x^2 + 2x + 1 =
     0, what is x?"}'

    _key_index = 0
    _key_lock = threading.Lock()

    @staticmethod
    def _load_wolfram_keys_from_env() ->
     List[str]:
        keys = []

        api_keys_str = os.getenv("
    WOLFRAM_ALPHA_API_KEYS", "").strip()
        if api_keys_str:
            for k in api_keys_str.split(
    ","):
                k = k.strip()
                if k and k not in keys:
                    keys.append(k)
```

```
            return keys

        if not DOTENV_AVAILABLE:
            return keys

        project_root = Path(__file__).
resolve().parent.parent
        candidate_env_files = [
            project_root / "api" / "
utils" / "keys.env",
            project_root / "keys.env",
            project_root / ".env",
        ]
        for env_path in
candidate_env_files:
            if env_path.exists():
                load_dotenv(env_path,
override=False)
                api_keys_str = os.getenv
("WOLFRAM_ALPHA_API_KEYS", "").strip
()
                if api_keys_str:
                    for k in
api_keys_str.split(","):
                        k = k.strip()
                        if k and k not
in keys:
                            keys.append(
k)
                if keys:
                    break

        return keys

    @classmethod
    def _get_next_key(cls, keys: List[
str]) -> str:
        if not keys:
            return ""
        with cls._key_lock:
            key = keys[cls._key_index %
len(keys)]
            cls._key_index = (cls.
_key_index + 1) % len(keys)
            return key

    def __init__(self, cfg=None, use_zh=
False):
        super().__init__(cfg, use_zh)
        self.api_keys: List[str] = []
        self.max_retries = 3

        if cfg is not None:
            cfg_keys = cfg.get("
wolfram_api_keys", [])
            if isinstance(cfg_keys, str)
:
                cfg_keys = [k.strip()
for k in cfg_keys.split(",") if k.
strip()]
            elif isinstance(cfg_keys,
list):
                cfg_keys = [k.strip()
for k in cfg_keys if isinstance(k,
str) and k.strip()]
            self.api_keys = cfg_keys
            self.max_retries = cfg.get("
wolfram_max_retries", 3)

        if not self.api_keys:
            self.api_keys = self.
_load_wolfram_keys_from_env()

    @staticmethod
    def _looks_like_interpretation(text:
 str) -> bool:
        if not text:
            return True
        t = text.strip().lower()
        patterns = (
            "input interpretation",
            "solve ",
            " for x",
            " for y",
            " for z",
        )
        return any(p in t for p in
patterns)

    @staticmethod
    def _parse_xml_response(xml_text:
str) -> Dict:
        root = ET.fromstring(xml_text)
        success_raw = root.attrib.get("
success", "false")
        success = str(success_raw).lower
() == "true"
        if not success:
            return {"error": "Your
Wolfram query is invalid. Please try
 a new query."}

        answer = ""
        for pod in root.findall("pod"):
            title = pod.attrib.get("
title", "")
            if title == "Solution":
                subpods = pod.findall("
subpod")
                if subpods:
                    plaintext = subpods
[0].findtext("plaintext") or ""
                    answer = plaintext
            if title in {"Results", "
Solutions"}:
                subpods = pod.findall("
subpod")
                for i, sub in enumerate(
subpods):
                    text = sub.findtext(
"plaintext") or ""
                    answer += f"ans {i}:
 {text}\n"
                break

        if not answer:
            preferred_titles = {
                "Result",
                "Results",
                "Solution",
                "Solutions",
                "Exact result",
                "Decimal approximation",
            }
            for pod in root.findall("pod
"):
                title = pod.attrib.get("
title", "")
                if title not in
preferred_titles:
```

```
                    continue
                sub = pod.find("subpod")
                if sub is not None:
                    text = (sub.findtext
("plaintext") or "").strip()
                    if text and not
SolveMathEquationTool.
_looks_like_interpretation(text):
                        answer = text
                        break

    if not answer or
SolveMathEquationTool.
_looks_like_interpretation(answer):
        return {"error": "No good
Wolfram Alpha result was found."}
    return {"result": answer.strip()
}

@staticmethod
def _is_retryable_error(error_msg:
str) -> bool:
    error_lower = error_msg.lower()
    retryable_patterns = ["rate
limit", "quota", "too many requests"
, "403", "429"]
    return any(p in error_lower for
p in retryable_patterns)

def _query_via_http(self, query: str
, api_key: str) -> Dict:
    try:
        resp = requests.get(
            "https://api.
wolframalpha.com/v2/query",
            params={"appid": api_key
, "input": query},
            timeout=30,
        )
        if resp.status_code != 200:
            return {
                "error": (
                    f"WolframAlpha
HTTP error: code={resp.status_code},
 "
                    f"body_preview={
resp.text[:240]}"
                ),
                "_retryable": resp.
status_code in (403, 429),
            }
        return self.
_parse_xml_response(resp.text)
    except Exception as e:
        return {
            "error": f"HTTP request
failed: {type(e).__name__}: {e}",
            "_retryable": self.
_is_retryable_error(str(e)),
        }

def _query_with_retry(self, query:
str) -> Dict:
    tried_keys = set()
    last_error = None

    for _ in range(min(self.
max_retries, len(self.api_keys))):
        current_key = self.
_get_next_key(self.api_keys)
        if current_key in tried_keys
:
            continue
        tried_keys.add(current_key)

        result = self.
_query_via_http(query, current_key)
        if "error" not in result or
not result.pop("_retryable", False):
            return result
        last_error = result

    if last_error:
        last_error.pop("_retryable",
 None)
    return last_error or {"error": "
All API keys exhausted."}

def call(self, params: Union[str,
Dict]) -> Dict:
    params_dict = self.parse_params(
params)
    query = params_dict["query"].
strip()

    if not query:
        return {"error": "Query
cannot be empty."}

    if not self.api_keys:
        return {
            "error": (
                "
WOLFRAM_ALPHA_API_KEYS is not set. "
                "Set it in config or
 environment (.env / keys.env)."
            )
        }

    return self._query_with_retry(
query)
```

Web Search. This tool queries external search engines using provided text keywords to retrieve up-to-date factual information. It returns the top relevant text snippets to supplement the internal knowledge of the agent.

```
class WebSearchTool(BasicTool):
    name = "web_search"
    description_en = (
        "Search the web for information
using a text query. "
        "Returns the top-k results, each
 with a title and snippet."
    )

    parameters = {
        "type": "object",
        "properties": {
            "query": {
                "type": "string",
                "description": "Search
query text",
            },
            "top_k": {
                "type": "integer",
                "description": "Number
of results to return (1-5, default:
```

```
3)",
            },
        },
        "required": ["query"],
    }
    example = '{"query": "Which city is the capital of China?", "top_k": 3}'

    def __init__(self, cfg: Optional[Dict] = None, use_zh: bool = False):
        super().__init__(cfg, use_zh)
        self._max_retries = 3
        self._retry_delay = 2

    def call(self, params: Union[str, Dict]) -> Dict:
        params_dict = self.parse_params(params)
        query = params_dict.get("query", "").strip()
        top_k = max(1, min(int(params_dict.get("top_k", 3)), 5))

        if not query:
            return {"error": "Parameter 'query' is required"}

        try:
            return self._text_search(query, top_k)
        except Exception as e:
            logging.error(f"Web search failed: {e}")
            return {"error": f"Search failed: {str(e)}"}

    async def call_async(self, params: Union[str, Dict]) -> Dict:
        return await asyncio.to_thread(self.call, params)

    def _text_search(self, query: str, top_k: int) -> Dict:
        url = "https://google.serper.dev/search"
        headers = {
            "X-API-KEY": GOOGLE_SEARCH_API_KEY,
            "Content-Type": "application/json",
        }
        payload = {
            "q": query,
            "gl": "cn",
            "hl": "zh-cn",
            "location": "China",
            "num": top_k,
        }

        for attempt in range(self._max_retries):
            try:
                resp = requests.post(
                    url, headers=headers, data=json.dumps(payload), timeout=30
                )
                resp.raise_for_status()
                data = resp.json()
                results = data.get("organic", [])[:top_k]

                if not results:
                    return {"search results": "No results found"}

                lines = []
                for i, res in enumerate(results, 1):
                    title = res.get("title", "")
                    snippet = res.get("snippet", "")
                    lines.append(f"[{i}] Title: {title}")
                    lines.append(f"Snippet: {snippet}")
                    lines.append("")

                return {"search results": "\n".join(lines).strip()}
            except Exception as e:
                logging.warning(
                    f"Request failed (attempt {attempt + 1}/{self._max_retries}): {e}"
                )
                if attempt < self._max_retries - 1:
                    time.sleep(self._retry_delay)
                else:
                    raise
```

E Efficiency Analysis

This section provides a detailed breakdown of the online and offline costs introduced by stateful experience retrieval. Throughout, we use Qwen3-VL-32B-Instruct as the base model and Qwen3-VL-8B-Embedding as the retrieval encoder, both served via vLLM on a single node with eight H800 GPUs, with 10-way client concurrency for both offline bank construction and online evaluation. Table 5 summarizes the per-benchmark efficiency profile.

Online retrieval overhead. The added per-query cost of MuSEAgent over a tool-using ReAct baseline comes entirely from the Deep-and-Wide search and not from extra base-model invocations: the per-step retrieval-view selection is produced jointly with the action plan within a single base-model call. At each retrieval-active step, the agent issues at most $L = 3$ viewpoint embeddings under the default configuration (Sec. 4.2), and embedding inference under vLLM is more than an order of magnitude faster than a base-model call on the same hardware, so retrieval latency is dominated by these embeddings rather than by base inference.

Table 5: Per-benchmark efficiency profile of MuSEAgent with Qwen3-VL-32B-Instruct on 8×H800 under vLLM (10-way client concurrency). Bank build time uses GPT-4o as the hindsight reasoner (the default in Sec. 4.2); evaluation wall-clock is end-to-end including all tool calls and Deep-and-Wide search.

Metric	V* Bench	MME-RW-Lite	Zoom-Bench	HR-Bench	Total
Exploration trajectories	95	959	310	100	1,464
Bank size $\|\mathcal{B}\|$	355	3,352	1,008	401	5,116
Bank build time	5 min	48 min	16 min	5 min	74 min
Evaluation queries	96	960	311	100	1,467
Avg. agent steps / query	6.6	9.1	7.8	7.9	—
Evaluation wall-clock	6 min	4h 37m	33 min	32 min	5h 48m

Table 6: Sensitivity of MuSEAgent (Qwen3-VL-32B-Instruct) to the quality score threshold δ. We report the accuracy (%) for all tasks. Best results are highlighted in **bold**.

Method	Exp. Level	Quality Score Threshold δ	V* Bench			HR-Bench
			Direct Attr.	Rel. Pos.	Micro Avg.	
w/ MuSEAgent	State	$\delta = 9.0$	72.41	78.95	75.00	73.00
w/ MuSEAgent	State	$\delta = 7.0$	77.59	78.95	78.12	74.00
w/ MuSEAgent	State	$\delta = 5.0$	**86.21**	**84.21**	**85.42**	**78.00**

Bank similarity search runs in-memory across the $M = 15$ viewpoint indices via batched FP16 cosine matmul on a single GPU and contributes a negligible fraction of step latency. The injected payload is at most $L \cdot K = 9$ short guidance snippets, bounded by L and K rather than by $|\mathcal{B}|$, so neither retrieval latency nor token overhead grows as more exploration trajectories are added.

Offline experience construction. The offline cost is incurred once per exploration trajectory and is fully amortized across all subsequent evaluation queries. The 1,464 exploration trajectories collected from Table 1 decompose into roughly 8,800 atomic transitions, of which 5,116 pass the $\delta = 5.0$ admission threshold and populate the four per-benchmark banks (Table 5). Each atomic transition is evaluated by $\mathcal{Q}_\phi$ to produce its quality score and guidance summary, conditioned on the realized trajectory outcome. Under the default setting in Sec. 4.2 where GPT-4o serves as the hindsight reasoner, building all four banks takes 74 minutes of wall-clock time and totals around $20 in API spend at standard GPT-4o rates, after which the same banks are reused across all subsequent evaluation queries. Under the *self-hindsight* setting (Sec. 4.5), where Qwen3-VL-32B-Instruct itself serves as the hindsight reasoner, this monetary cost is replaced by additional vLLM inference time on the same in-house hardware; either path produces a comparable bank without further offline cost.

Online evaluation cost is benchmark-dominated. Across the four benchmarks (Table 5), the per-benchmark wall-clock spans from 6 minutes (V* Bench, 96 queries) to 4 hours 37 minutes (MME-RealWorld-Lite, 960 queries). The roughly 46× gap mirrors the corresponding ∼10× growth in evaluation queries combined with the ∼1.4× increase in average agent steps per query, rather than any retrieval-side scaling: average steps per query stay within 6.6–9.1 across all benchmarks, and the bank similarity search remains in-memory regardless of $|\mathcal{B}|$. The dominant per-query cost is therefore base-model inference, not Deep-and-Wide search.

Scaling behavior. Because L and K are fixed configuration parameters, both the per-query retrieval latency and the per-query token overhead remain flat as the experience bank grows. Adding more exploration trajectories therefore enlarges the offline construction cost linearly while leaving online inference cost unchanged, which makes the offline cost increasingly amortizable as the bank is queried by more deployment traffic.

F Quality Score Threshold Sensitivity

The quality threshold δ controls the admission criterion of the experience bank: only transitions whose hindsight score $q_t \geq \delta$ are retained, so

larger δ shrinks the bank toward near-optimal decisions while smaller δ admits a broader range of partially correct or instructive sub-decisions. We sweep $\delta \in \{9.0, 7.0, 5.0\}$ with Qwen3-VL-32B-Instruct on V* Bench and HR-Bench, holding all other components at their defaults so the effect is isolated to bank composition.

A moderately permissive threshold is uniformly better. Table 6 shows accuracy increases monotonically as δ relaxes: V* Bench moves from 75.00% at $\delta = 9.0$ to 78.12% at $\delta = 7.0$ and 85.42% at $\delta = 5.0$, with HR-Bench tracking the same trend (73.00 / 74.00 / 78.00%). The trajectory is consistent across both benchmarks and across the two V* Bench sub-tasks, indicating that the trend reflects a property of the bank rather than a per-dataset artifact.

Strict filtering removes the very experiences hindsight is meant to surface. A high δ keeps only canonical optimal moves, but stateful experience derives much of its value from *near-misses, corrective steps, and erroneous tool invocations whose hindsight guidance encodes what to avoid*, all of which receive intermediate rather than top scores. Discarding them collapses the bank into a small set of textbook trajectories that rarely match the diverse decision states encountered at inference time, so retrieval more often hits the empty-set fallback (Sec. A) and the agent reverts to its ReAct-style policy without cautionary signals about misused tools. Conversely, $\delta = 5.0$ retains transitions that $\mathcal{Q}_\phi$ judged at least adequate (the midpoint of its $[0, 10]$ scale), which is permissive enough to preserve corrective guidance and tool-failure warnings while still excluding clear no-ops and uninformative side moves. The robustness of accuracy across the swept range further suggests that exact tuning of δ is not required, the design is forgiving as long as the threshold sits in the lower half of the score range.

G Case Study

Figure 3 traces a representative V* Bench instance end-to-end, showing how the abstraction phase populates the bank and the exploitation phase consumes it on a structurally similar query.

Offline: hindsight scoring turns one trajectory into differentiated experiences. The left panel records a fine-grained attribute recognition task where the agent runs object localization, image cropping, OCR, and text-image similarity matching to identify the color of a specified umbrella. Instead of storing this trajectory as a single retrievable unit, MuSEAgent evaluates each state-action pair independently: the initial localization is scored 9 because attribute reasoning is impossible without first isolating the target region, while OCR on the cropped region is scored 1 because text extraction is irrelevant for a purely visual color attribute. The two scores produce two qualitatively different experiences, one prescribing localization-before-attribute-analysis and one warning against OCR on visual-only queries, that would be entangled and washed out under trajectory-level retrieval.

Online: multi-viewpoint retrieval converges on a consistent action. On a similar query, the right panel shows the agent decomposing its current state into distinct semantic viewpoints (e.g., query with visual input, query with task description) and querying the bank under each. Different viewpoints surface different past experiences, but their natural-language guidance independently agrees on the same prescription, spatial localization before detailed visual analysis, which the agent then commits to as its action plan. This convergence across viewpoints is exactly the signal the union-based aggregation in Algorithm 2 is designed to amplify: agreement under heterogeneous queries is stronger evidence than a single high-similarity match.

What the case validates. The two panels concretize three design choices: (i) hindsight scoring attaches differentiated quality to atomic decisions rather than caching monolithic trajectories; (ii) low scores on ineffective tool calls (e.g., OCR scoring 1) preserve cautionary signals that trajectory-level retrieval would absorb as noise; and (iii) Deep-and-Wide search leverages multi-viewpoint convergence to remain robust against the noisy, non-deterministic outputs typical of multimodal tool use.

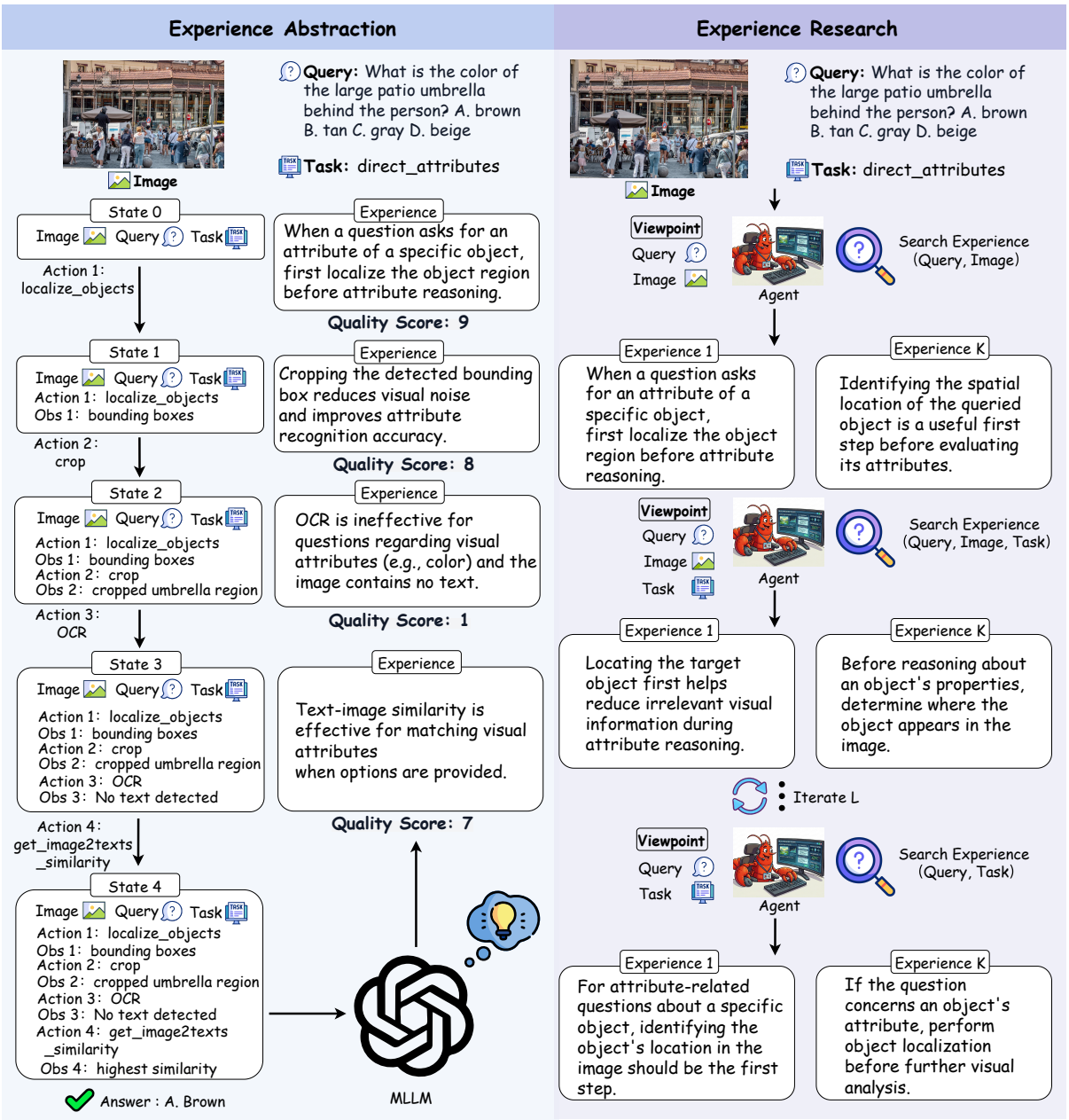


Figure 3: A case of stateful experience abstraction and exploitation.

H Prompts

H.1 Evaluation Prompt

Evaluation Prompt

Role: You are a multimodal assistant with tool access and memory.

Task: Analyze images and questions carefully, decide whether tool usage is necessary, and use tools strategically when they help obtain information that cannot be reliably determined through direct observation.

Tool Usage Guidelines:
Before using tools, analyze and plan:

- Carefully analyze the task requirements and image content
- Identify what information is required to answer the question
- Evaluate which tools are suitable for the current task
- Plan the sequence of tool usage if multiple tools are needed
- Consider how outputs from multiple tools can be combined to derive the final answer

When tools are appropriate:

- When examining details that are difficult to see directly (e.g., small text, tiny objects, subtle textures)
- When precise localization or counting is required
- When the image content is complex and requires step-by-step regional analysis

When tools are not appropriate:

- When cropping or zooming would remove important contextual information
- When the answer can already be determined through direct observation

Best practices when using tools:

- Clear objectives: know exactly what information you want before calling a tool
- Accurate regions: ensure the selected region corresponds to the part that requires analysis
- Analyze outputs critically and combine them with direct observation
- Do not replace visible evidence with speculative inference

If a tool call fails or returns an error, adapt by trying alternative tools or adjusting your strategy.

Image Reference:
All images are referenced with unique IDs (img_0, img_1, ...). When calling tools, use these IDs to specify which image to operate on.

[If stateful experience search is enabled]:
Experience Retrieval:
Before each action step, you MUST call `search_experiences` to retrieve advice from similar previously solved tasks.

- Choose a retrieval view that best matches the current information need
- Multiple retrieval calls are allowed if necessary
- Use retrieved advice to guide decision making before taking the next action

H.2 Prompt for Stateful Experience Abstraction

Prompt for Stateful Experience Abstraction

Role: You are evaluating a complete agent reasoning trace.
[If the task was solved CORRECTLY]: The task was solved CORRECTLY. The process may be highly efficient, or it may contain wasteful, redundant, or error-prone steps. Judge each step by the quality and relevance of its output.
[If the task was solved INCORRECTLY]: The task was solved INCORRECTLY. Your goal is to identify which steps were most responsible for the incorrect outcome. A wrong final answer does NOT mean every step was bad —

many steps may have been perfectly reasonable. Focus on pinpointing the critical steps that introduced errors, ignored evidence, or led the reasoning astray.

Task:
{images_note}
Question: {question} {type_line}
{tools_section}

Trajectory:
The initial state s_0 is the question and images above. Each subsequent state is what the agent has seen so far, including all previous actions and observations.
{traj_text}

Instructions:
[If the task was solved CORRECTLY]: For each (state, action) pair, provide:
1. q_value (0-10): Rate the quality of this action at this state. Judge whether it produced accurate, relevant information that actually contributed to reaching the answer.

- **9-10 (Essential):** Produced decisive information that directly shaped the answer, or answered correctly at the right time
- **7-8 (Helpful):** Useful output that advanced the solution, with minor room for improvement in tool choice or parameters
- **5-6 (Reasonable):** Valid approach, but output had little actual influence on the final answer; a more direct path existed
- **3-4 (Wasteful):** Produced no new information, duplicated prior knowledge, or was an unnecessary detour
- **0-2 (Harmful):** Produced errors, misleading output, or repeated a known failure

[If the task was solved INCORRECTLY]: Rate how much this step CONTRIBUTED TO the incorrect outcome. A high score means this step was a critical cause of the error.

- **9-10 (Critical error):** Directly caused or cemented the wrong answer (e.g., misread evidence, wrong tool, flawed conclusion)
- **7-8 (Significant):** Substantially misled the reasoning, even if not the sole cause of failure
- **5-6 (Moderate):** Somewhat contributed to the error, but other steps were more decisive
- **3-4 (Minor):** Had little impact on the incorrect outcome; the step was mostly neutral
- **0-2 (Irrelevant):** This step was reasonable and did not contribute to the failure at all

2. experience (1-2 sentences): Actionable advice that a FUTURE agent would see BEFORE making this decision. This advice will be injected into the agent's context at runtime to guide its next action.
Guidelines:

- Reference the task type and tool strategy, but keep the advice generalizable — do not mention specific objects, labels, or values from this particular trace.
- NEVER mention or hint at the correct answer.
- **[If the task was solved CORRECTLY]:** Focus on STRATEGY: recommend the specific tool and approach that works, or advise answering directly if tools are unnecessary. If the step was an error, redundant, or unnecessary, the experience MUST be cautionary.
- **[If the task was solved INCORRECTLY]:** For high `q_value` steps (critical errors), the experience MUST be a cautionary warning — explain what went wrong and what to avoid. For low `q_value` steps, provide brief neutral or positive guidance.

Examples:

- **Good (Strategy):** "For visual similarity tasks, use `get_image2images_similarity` to get objective scores rather than estimating visually."
- **Good (Direct Answer):** "The observation already contains enough information to answer directly. No further tool calls are needed."
- **Good (Correction):** "When observations conflict with prior reasoning, re-examine the evidence before committing to an answer."
- **Bad (Too Vague):** "The agent should use the right tool." / "The agent should be more careful."
- **Bad (Leaks Answer):** "The answer is likely A based on the scores."

Output Format:
Output JSON array:
```
[{"state": 0, "q_value": 8, "experience": "..."}, ...]
```

H.3 Prompt for Trajectory-level Experience Abstraction

Prompt for Trajectory-level Experience Abstraction

Role: You are evaluating a complete agent reasoning trace.

[If the task was solved CORRECTLY]: The task was solved CORRECTLY. Your goal is to summarize what strategy worked well across the reasoning trajectory. Focus on effective tool usage, evidence interpretation, and reasoning strategies that helped reach the correct answer.
[If the task was solved INCORRECTLY]: The task was solved INCORRECTLY. Your goal is to identify what went wrong in the reasoning trajectory and what a future agent should do differently to avoid similar mistakes.

Task:
{images_note}
Question: {question} {type_line}
{tools_section}

Trajectory:
{traj_text}

Instructions:
Write exactly **1–2 concise sentences** of actionable experience for future agents facing similar tasks.
[If the task was solved CORRECTLY]:

- Summarize the strategy that worked well (e.g., tool choice, evidence usage, reasoning approach)
- Highlight patterns that helped reach the correct answer efficiently
- Focus on practices that future agents should replicate

[If the task was solved INCORRECTLY]:

- Identify what went wrong in tool usage, reasoning, or evidence interpretation
- Explain what a future agent should do differently
- Highlight common mistakes or misleading reasoning patterns to avoid

General Guidelines:

- Keep the advice generalizable to similar tasks
- Do not mention specific objects, labels, or values from the trajectory
- NEVER reveal or hint at the correct answer

Output Format:
Output JSON object:
```
{"experience": "your 1-2 sentence experience"}
```