Mem²Evolve: Towards Self-Evolving Agents via Co-Evolutionary Capability Expansion and Experience Distillation

Zihao Cheng[1], Zeming Liu[1†], Yingyu Shan[2], Xinyi Wang[3], Xiangrong Zhu[3], Yunpu Ma[4,6], Hongru Wang[5], Yuhang Guo[2], Wei Lin[3], Yunhong Wang[1],
[1]Beihang University [2]Beijing Institute of Technology [3]Independent Researcher
[4]Ludwig Maximilian University of Munich [5]University of Edinburgh
[6]Munich Center for Machine Learning
[†]Corresponding author Email: zihaocheng@buaa.edu.cn, zmliu@buaa.edu.cn

Abstract

While large language model–powered agents can self-evolve by accumulating experience or by dynamically creating new assets (i.e., tools or expert agents), existing frameworks typically treat these two evolutionary processes in isolation. This separation overlooks their intrinsic interdependence: the former is inherently bounded by a manually predefined static toolset, while the latter generates new assets from scratch without experiential guidance, leading to limited capability growth and unstable evolution. To address this limitation, we introduce a novel paradigm of co-evolutionary Capability Expansion and Experience Distillation. Guided by this paradigm, we propose the **Mem²Evolve**, which integrates two core components: **Experience Memory** and **Asset Memory**. Specifically, Mem²Evolve leverages accumulated experience to guide the dynamic creation of assets, thereby expanding the agent's capability space while simultaneously acquiring new experience to achieve co-evolution. Extensive experiments across 6 task categories and 8 benchmarks demonstrate that Mem²Evolve achieves improvement of 18.53% over standard LLMs, 11.80% over agents evolving solely through experience, and 6.46% over those evolving solely through asset creation, establishing it as a substantially more effective and stable self-evolving agent framework. Code is available at: https://buaa-irip-llm.github.io/Mem2Evolve.

1 Introduction

Large language model (LLM)–powered agents have achieved remarkable success in a wide range of applications (Liu et al., 2020; Yang et al., 2024; Jin et al., 2025; Deng et al., 2025; Liu et al., 2025a; Cheng et al., 2025b). Building on these successes, recent research is moving beyond static, task-specific systems toward self-evolving agents that can leverage past experiences and autonomously expand their capabilities (Gao et al., 2025; Fang et al., 2025; Wang et al., 2025a; Han et al., 2026; Jiang et al., 2025; Liu et al., 2025b).

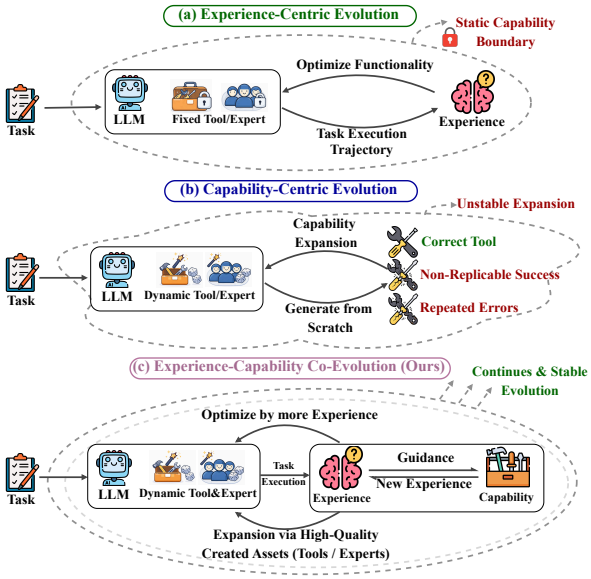


Figure 1: **Paradigms of Self-Evolving Agents**: (a) Experience-centric evolution, (b) Capability-centric evolution, and (c) Our co-evolutionary framework that jointly expands capabilities and distills experience.

However, current frameworks predominantly treat these evolutionary processes in isolation (Cemri et al., 2025). As illustrated in Figure 1(a), **Experience-centric evolution** (Yuan et al., 2025; Yuksekgonul et al., 2025) enables systems to learn from experience to optimize execution strategies (Ma et al., 2025), refine prompts (Zhang et al., 2025a), or build experience repositories (Ouyang et al., 2025). However, this paradigm limits the system to a fixed set of tools and expert agents, leading capability space remains static and cannot expand beyond the pre-specified library. In contrast, **capability-centric evolution** (Figure 1(b)) enables the system to dynamically create new tools (Wölflein et al., 2025; Qiu et al., 2025) or spawn new expert agents (Chen et al., 2023, 2024; Zhang et al., 2025b). However, creat-

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 20784–20831
July 2-7, 2026 ©2026 Association for Computational Linguistics

Framework	Experience Distillation			Capability Expansion				Exp.-Guided Creation
	Optimization	Persistence	Source	Tool Crea.	Agent Crea.	Tool/Agent	Crea. Grounding	
DSPy (Khattab et al., 2023)	✓	✗		✗	✗	Static	–	✗
DyLAN (Liu et al., 2023)	✓	✗		✗	✗	Static	–	✗
ReasoningBank (Ouyang et al., 2025)	✗	✓		✗	✗	Static	–	✗
AFlow (Zhang et al., 2025a)	✓	✗		✗	✗	Static	–	✗
AgentSquare (Shang et al., 2025)	✓	✗		✗	✗	Static	–	✗
Agentic Neural Networks (Ma et al., 2025)	✓	✗		✗	✗	Static	–	✗
AgentVerse (Chen et al., 2023)	✓	✗	–	✗	✓	Dynamic		✗
AutoAgents (Chen et al., 2024)	✗	✗	–	✗	✓	Dynamic		✗
SwarmAgentic (Zhang et al., 2025b)	✓	✗	–	✗	✓	Dynamic		✗
Alita (Qiu et al., 2025)	✗	✗	–	✓	✗	Dynamic	+	✗
ToolMaker (Wölflein et al., 2025)	✗	✗	–	✓	✗	Dynamic	+	✗
Mem²Evolve (Ours)	✓	✓	+	✓	✓	**Dynamic**	+ +	✓

Table 1: **Comparison of self-evolving agent frameworks. Optimization** indicates whether experience is used to optimize the agent (e.g., prompts). **Persistence** denotes whether experiences are persistently stored for future reuse. **Source**: agent task execution trajectory, tool creation process. **Tool Crea.** and **Agent Crea.** indicate whether the framework supports creation of tools and expert agents, respectively. **Tool/Agent** denotes whether the toolset and expert agents are static or dynamic. **Crea. Grounding** indicates the knowledge sources used for asset creation, parametric knowledge, web search information, experience. **Exp.-Guided Creation** indicates whether new assets are created under the guidance of past experience. Details in the Appendix A.1 and A.2.

ing new assets from scratch without the guidance of experience prevents the system from utilizing proven strategies and avoiding known pitfalls, leading to non-replicable success and repeated errors.

To address these limitations, inspired by the equilibrium theory (Piaget, 1972), which posits intelligence evolves through the interplay of assimilation (integrating new experiences) and accommodation (adapting internal structures), we introduce a novel paradigm of *co-evolutionary capability expansion and experience distillation* (Figure 1c). In this paradigm, expanding agent capabilities enables it to complete a broader range of tasks, thereby yielding more experiences. These experiences are then distilled to guide subsequent capability expansion, realizing co-evolution of capability and experience.

Guided by this paradigm, we propose **Mem²Evolve**, an agentic framework that coordinates the evolution of capabilities and experiences through a core dual-memory mechanism comprising Asset Memory and Experience Memory. Specifically, Asset Memory serves as a persistent and extensible repository of the agent's capabilities, organizing expert agents and executable tools. Experience Memory accumulates strategic experience distilled from both successful and failed trajectories to guide future asset creation and task execution. Building upon this dual-memory architecture, Mem²Evolve operates through two complementary phases. During *forward inference*, the system follows a "reuse first, create on demand" strategy, leveraging both memories to execute tasks. When a task exceeds the agent's current capability boundary, the system dynamically creates new assets guided by experience to expand its capabilities. Upon task completion, *backward evolution* retains high-quality newly created assets into Asset Memory and distills transferable lessons into Experience Memory. This forward-backward loop enables the co-evolution of capabilities and experiences.

To validate the effectiveness of Mem²Evolve, we conduct extensive experiments across 6 tasks and 8 benchmarks, covering general assistant (Mialon et al., 2024), multi-hop question answering (Yang et al., 2018), mathematical reasoning, embodied task (Shridhar et al., 2020), planning (Xie et al., 2024), and web interaction (Yao et al., 2022a). Beyond achieving superior overall performance against capability- and experience-centric baselines, Mem²Evolve demonstrates robust adaptability, enabling sustained evolution in *single-task* and effective memory reuse in *cross-task* settings.

Our contributions are summarized as follows:

- To the best of our knowledge, we are the first to propose the co-evolutionary agent paradigm that couples dynamic capability expansion with experience distillation.
- Guided by this paradigm, we introduce Mem²Evolve, a dual-memory framework that coordinates Asset Memory for dynamic capability expansion and Experience Memory for strategic experience distillation. Through a forward inference and backward evolution loop, Mem²Evolve continuously leverages and expands both memories, driving capability–experience co-evolution.

- Extensive experiments show that Mem^{2}Evolve consistently outperforms both capability-centric and experience-centric baselines. Moreover, it exhibits strong adaptability, supporting sustained self-evolution in single-task settings and effective memory reuse for cross-task generalization.

2 Related Work

Experience-Centric Evolving. Recent research on self-evolving agents predominantly focuses on optimizing systems by leveraging experience accumulated from past tasks (Yuan et al., 2025; Li et al., 2023; Ma et al., 2025; Hao et al., 2026; Bai et al., 2026). For instance, DyLAN (Liu et al., 2023) and DSPy (Khattab et al., 2023) dynamically select agent teams from a predefined pool by aligning past experience with current task requirements. Similarly, Aflow (Zhang et al., 2025a) and AgentSquare (Shang et al., 2025) modularize agents and employ search algorithms to optimize module compositions. ReasoningBank (Ouyang et al., 2025) summarizes successful and failed experiences to enhance performance on new tasks. However, as shown in Table 1, these frameworks are confined to a fixed set of tools and agents, resulting in a static capability space. Consequently, they cannot extend their boundaries to handle tasks beyond the predefined asset. In contrast, Mem^{2}Evolve dynamically creates high-quality agents and tools, enabling it to transcend these pre-existing capability limits.

Capability-Centric Evolving. In parallel to experience-centric evolving, capability-centric frameworks focus on expanding the boundaries of agentic systems by dynamically generating tools or agents, thereby reducing dependence on manual design (Cai et al., 2024; Song et al., 2024; Long et al., 2026; Qiyuan et al., 2026). AgentVerse (Chen et al., 2023) and AutoAgents (Chen et al., 2024) generate expert agents tailored to specific task dynamics, extending the system's execution capabilities. ToolMaker (Wölflein et al., 2025) and Alita (Qiu et al., 2025) dynamically create tools to handle videos, documents, and complex mathematical simulations (Feng et al., 2025; Wan et al., 2026). However, creating these new assets from scratch without the guidance of experience prevents these systems from leveraging proven strategies or avoiding known pitfalls. This isolation inevitably leads to non-replicable successes and recurring errors. In contrast, Mem^{2}Evolve couples capability expansion with experience distillation, realizing a co-evolution that past insights guide asset creation and new capabilities yield richer experiences.

3 Mem^{2}Evolve

We present **Mem^{2}Evolve**, a novel self-evolving agent framework that coordinates capability expansion and experience distillation. As illustrated in Figure 2, Mem^{2}Evolve is built upon a *Dual-Memory Mechanism*: *Asset Memory* for dynamic capability expansion and *Experience Memory* for strategic experience distillation(§3.1). Built on this dual-memory foundation, Mem^{2}Evolve operates in a two-phase task loop: *forward inference* and *backward evolution*. During *forward inference* (§3.2), the agent leverages both memories to execute tasks while dynamically creating new assets to expand its capabilities boundary. Upon task completion, the *backward evolution* process (§3.3) retains high-quality assets and distills lessons from execution trajectories, enabling continuous self-evolution.

3.1 Dual-Memory Mechanism

We organize the memory into two distinct components: the **Asset Memory** $\mathcal{M}_A$, which stores the expert agents and tools, and the **Experience Memory** $\mathcal{M}_E$, which accumulates lessons distilled from past successes and failures to guide future actions.

3.1.1 Asset Memory

To support capability expansion at both the strategic level (through expert agents) and the operational level (through tools), Asset Memory serves as a repository of reusable, execution-ready capabilities:

$$\mathcal{M}_A = \mathcal{B}_{agt} \cup \mathcal{B}_{tool}, \tag{1}$$

where $\mathcal{B}_{agt}$ is the *Agent Bank* containing expert agents, and $\mathcal{B}_{tool}$ is the *Tool Bank* that stores executable tools.

Agent Bank. Building on prior work (Chen et al., 2024) and Anthropic's Agent Skills[1], we distill a compact *agent specification* tailored to Mem^{2}Evolve. As exemplified in Figure 6, each entry $m_{agt} \in \mathcal{B}_{agt}$ is defined as:

$$m_{agt} = \langle \rho, \epsilon, \sigma, \mathbb{T}_{avail} \rangle, \tag{2}$$

where ρ is the *role* specifying the agent's identity, ϵ describes the agent's *expertise* and domain knowledge, σ denotes *suggestions* that guide the agent's

[1] https://github.com/anthropics/skills

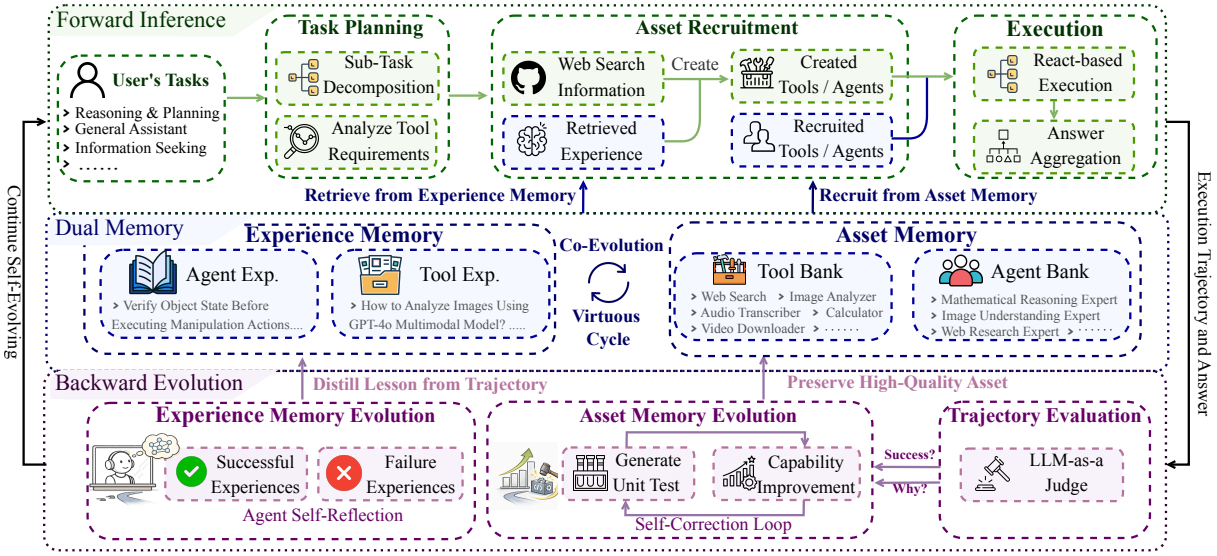


Figure 2: **Overview of Mem²Evolve**, a self-evolving agent framework built on a **Dual-Memory** mechanism. The evolution proceeds in two phases. During **Forward Inference**, the agent recruits tools and expert agents from *Asset Memory* to execute the current task. When the task exceeds its current capability boundary, *Experience Memory* is leveraged to guide the stable creation of new assets on demand. During **Backward Evolution**, newly validated assets are preserved in Asset Memory to achieve persistent capability expansion, while strategic insights distilled from execution trajectories are accumulated into Experience Memory. This forward–backward loop enables the co-evolution of capabilities and experience, forming a stable self-evolving cycle.

behavior strategies, and $\mathbb{T}_{avail} \subseteq \mathcal{B}_{tool}$ specifies the set of available tools.

Tool Bank. To ensure seamless integration with diverse LLM backbones, Tool Bank maintains executable tools stored in compliance with the Model Context Protocol (MCP)[2], example in Code 1. Each entry $m_{tool} \in \mathcal{B}_{tool}$ is defined as:

$$m_{tool} = \langle n, d_{func}, c_{impl}, \omega_{doc} \rangle, \tag{3}$$

where n is the *tool name*, d_{func} provides a *functional description*, c_{impl} contains the *implementation code*, and ω_{doc} specifies input/output documentation.

3.1.2 Experience Memory

To enable Mem²Evolve to replicate proven strategies and circumvent previously encountered pitfalls, Experience Memory accumulates insights derived from past successes and failures, guiding future task execution and asset creation (Ouyang et al., 2025). We define $\mathcal{M}_E = \mathcal{E}_{agt} \cup \mathcal{E}_{tool}$, with each *Memory Item* $e \in \mathcal{M}_E$ structured as:

$$e = \langle h_{title}, d_{desc}, \mathcal{U}_{case}, \kappa_{content} \rangle, \tag{4}$$

where h_{title} is the title, d_{desc} describes the context, $\mathcal{U}_{case}$ lists applicable use cases, and $\kappa_{content}$ stores the core distilled knowledge, encompassing both agent experience and tool experience:

Agent Experience. $\kappa_{content}$ contains strategic insights derived from trajectory reflections, guiding specific expert agents in handling complex tasks.

Tool Experience. $\kappa_{content}$ contains implementation guidelines distilled from the tool creation and debugging process, and an example in Figure 8.

3.2 Forward Inference

To balance the utilization of accumulated expertise with the acquisition of new capabilities, the forward inference follows a strategy of *"Reuse first, Create on demand"*. We formalize it into three phases: *(1) task planning*, *(2) asset recruitment*, and *(3) execution*.

3.2.1 Task Planning

Initially, the LLM π_θ acts as a planner to decompose the task q_t into a sequence of sub-tasks $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$, with the prompt in Appendix C.1. This decomposition ensures that complex problems are broken down into solvable units with clear resource definitions.

[2] https://www.anthropic.com/news/model-context-protocol

3.2.2 Asset Recruitment

For each sub-task s_i, the system prepares the required assets via the Recruitment Function $\Gamma(s_i)$:

$$\Gamma(s_i) = \begin{cases} m^* & \text{sim}(s_i, \mathcal{M}_A) \geq \delta \\ \text{Create}(s_i \mid \mathcal{M}_E, \text{Web}) & \text{otherwise} \end{cases} \quad (5)$$

where $\text{sim}(s_i, \mathcal{M}_A)$ measures the similarity between the sub-task and the asset stored in Assets Memory, and δ is a confidence threshold. This mechanism determines whether the sub-task lies beyond the agent's current capability boundary. Depending on the output of $\Gamma(s_i)$, the process branches into two paths:

Recruitment. If a high-similarity match exists, the system directly reuses $m^* \in \mathcal{M}_A$. For agents, we select the top-1 candidate surpassing δ to entrust the sub-task to the most specialized expert. Conversely, for tools, we retrieve the top-k matches to ensure comprehensive utility while mitigating context overhead from excessive documentation.

Creation. Conversely, for missing capabilities, *Tool Creation* employs experience-augmented generation, conditioning on the sub-task s_i, web search results, and relevant experiences e from $\mathcal{E}_{tool}$:

$$m_{tool}^{new} \sim \pi_\theta(s_i \mid \text{Retrieve}(s_i, \mathcal{E}_{tool}), \text{Web}(s_i)). \quad (6)$$

Similarly, *Agent Creation* synthesizes a new expert by prompting π_θ with task requirements derived from s_i, and details in Appendix A.3.

3.2.3 Execution

Each sub-task s_i is assigned to its recruited agent m_{agt}^i, augmented with experiences e retrieved from $\mathcal{E}_{agt}$ for role-specific guidance. The agent then executes using available tools $\mathbb{T}_{avail}^i$ within the ReAct framework (Yao et al., 2022b), alternating among think, action, and observation steps. Finally, system aggregates all results $\{r_1, \ldots, r_k\}$ to produce the final answer a_t.

3.3 Backward Evolution

Upon task completion, the backward evolution aims to preserve high-quality assets for future reuse and distill transferable lessons from execution trajectories. We formalize it into: *(1) trajectory evaluation, (2) asset memory evolution, and (3) experience memory evolution.*

3.3.1 Trajectory Evaluation

The evaluation stage provides the foundation for all subsequent memory updates. We employ an LLM-as-a-Judge (Li et al., 2025a; Ouyang et al., 2025; Cheng et al., 2025a; Ma et al., 2026; Yue et al., 2025) to assess execution quality.[3] Given the task q_t, execution trajectory τ_t, and answer a_t, the Judge produces:

$$r_t, c_t = \text{Judge}(q_t, \tau_t, a_t), \quad (7)$$

where $r_t \in \{0, 1\}$ indicates success or failure, and c_t provides critique comments identifying specific strengths and weaknesses.

3.3.2 Asset Memory Evolution

This phase determines which newly created assets should be preserved and refined before entering $\mathcal{M}_A$. Since a correct answer does not guarantee robust underlying assets, we adopt a *Self-Correction Loop* guided by r_t and c_t.

For each asset $m_{\text{new}} \in \mathcal{A}_t^{new}$, where $\mathcal{A}_t^{new}$ denotes the set of newly created assets, we derive a finalized version m_{final} as:

$$m_{\text{final}} = \begin{cases} m_{\text{new}} & \text{if } r_t = 1 \wedge \\ & \text{Valid}(m_{\text{new}}, c_t) \\ \text{Improve}(m_{\text{new}}, c_t) & \text{otherwise} \end{cases} \quad (8)$$

Where $\text{Valid}(m_{\text{new}}, c_t)$ verifies asset reliability by having the LLM synthesize test cases from the critique c_t and executing m_{new} against them. An asset passes validation only if it clears all tests.

If validation fails, $\text{Improve}(m_{\text{new}}, c_t)$ triggers a *Self-Correction Loop*: revise the asset based on c_t and test failures, then regenerate tests until validation passes. Once validated:

$$\mathcal{M}_A \leftarrow \mathcal{M}_A \cup \{m_{\text{final}}\}. \quad (9)$$

3.3.3 Experience Memory Evolution

Mem^{2}Evolve distills trajectory-level insights into the $\mathcal{M}_E$ to guide future task execution and asset creation. After each task, the system reflects on the trajectory τ_t and (r_t, c_t) to extract *Memory Items*:

$$e_{\text{new}} = \text{Reflection}(\tau_t, r_t, c_t), \quad (10)$$

The Reflection function captures insights from both successful and failed executions.

Success Generalization. When $r_t = 1$, Mem^{2}Evolve abstracts high-level guidelines from the successful trajectory. For agents, $\kappa_{content}$ records strategic advice and coordination patterns for specific roles; for tools, it captures effective implementation patterns and usage recipes.

[3] We assume ground-truth labels are inaccessible during backward evolution to simulate real world. When available, such supervision can further enhance evolution effectiveness.

Method	GAIA				Embodied	Multi-Hop QA		Math		Planning	Web Interaction	Avg.
	L1	L2	L3	Total	ALFWorld	HotpotQA	2Wiki	AIME24	AIME25	TravelPlanner	WebShop	
Naive-Large Language Model												
GPT-5-Chat (Direct)	16.98	12.79	7.69	12.49	83.58	50.40	81.80	60.00	46.67	38.68	22.31	49.49
GPT-5-Chat (CoT)	24.53	17.44	11.54	17.84	83.58	47.40	74.40	66.67	56.67	39.51	27.49	51.71
GPT-5-Chat (ReAct)	26.42	17.44	11.54	18.47	86.87	41.40	48.40	66.67	60.00	39.13	25.10	48.27
OpenAI-DeepResearch†	74.29	69.06	47.60	67.36	—	—	—	—	—	—	—	—
Experience-Centric Evolving												
DyLAN	24.53	19.78	11.54	18.62	91.20	52.00	65.00	46.67	43.33	43.15	36.40	49.55
EvoAgent	22.64	19.78	11.54	17.99	92.50	54.40	75.00	66.67	43.33	49.20	37.80	54.61
AFLOW	26.42	17.44	15.38	19.75	93.40	**60.80**	72.40	66.67	63.33	53.24	37.90	58.44
DSPy	30.19	15.12	11.54	18.95	92.80	55.60	76.40	66.67	50.00	44.90	35.50	55.10
Capability-Centric Evolving												
Alita	81.13	75.58	46.15	72.73	86.13	58.80	77.40	70.00	66.67	48.32	30.21	63.78
AgentVerse	30.19	16.28	19.23	21.90	88.32	38.60	74.60	60.00	50.00	47.25	32.53	51.65
AutoAgents	35.85	24.42	19.23	26.50	87.92	54.20	73.80	40.00	36.67	43.52	31.40	49.25
SwarmAgentic	28.30	18.60	13.46	20.40	88.79	56.00	80.00	46.67	40.00	59.14	34.12	53.14
Ours												
Mem²Evolve	**88.68**	**82.56**	**57.69**	**76.31**	**94.31**	**60.80**	**82.00**	**76.70**	**73.33**	**59.25**	**39.20**	**70.24**

Table 2: **Main results across 6 tasks and 8 benchmarks**, reported as Pass@1 for each benchmark. The best results are highlighted in **bold**, and the second-best results are underlined. †Results are from the original paper.

Failure Diagnosis. When $r_t = 0$ or c_t indicates substantial debugging, Reflection focuses on failure modes. The resulting e_{new} encodes anti-patterns and failure–fix pairs to prevent similar errors. Detailed prompt in Appendix C.8 and C.9.

Finally, the distilled experience items are merged into the Experience Memory:

$$\mathcal{M}_E \leftarrow \mathcal{M}_E \cup \{e_{\text{new}}\}. \tag{11}$$

4 Experiments

4.1 Experiment Setting

Baselines. Following prior research (Qiu et al., 2025; Zhang et al., 2025b; Zhou et al., 2026), we compare Mem²Evolve against three categories of baselines: (1) **Naive LLMs**, including Direct prompting, CoT (Wei et al., 2022), ReAct (Yao et al., 2022b), and OpenAI's DeepResearch (OpenAI, 2025), (2) **Experience-Centric frameworks** such as DyLAN (Liu et al., 2023), EvoAgent (Yuan et al., 2025), AFLOW (Zhang et al., 2025a), and DSPy (Khattab et al., 2023), and (3) **Capacity-Centric frameworks**, spanning tool-generative methods Alita (Qiu et al., 2025), ToolMaker (Wölflein et al., 2025)) and agent-generative approaches AgentVerse (Chen et al., 2023), AutoAgents (Chen et al., 2024), SwarmAgentic (Zhang et al., 2025b). More details in the Appendix B.1.

Benchmarks. Following Li et al. (2025b); Wang et al. (2025b), we evaluate the agent's capabilities across 8 benchmarks in 6 distinct tasks. These include **GAIA** (Mialon et al., 2024) for general assistant, **ALFWorld** (Shridhar et al., 2020) and **WebShop** (Yao et al., 2022a) for embodied and web interaction, and **TravelPlanner** (Xie et al., 2024) for planning. We also include **HotpotQA** (Yang et al., 2018) and **2WikiMultihopQA** (Ho et al., 2020) for multi-hop QA, plus **AIME 24/25** for mathematical reasoning. Details are in Appendix B.2.

Implement Details. For all baselines, we utilize GPT-5-chat[4] as the LLM backbone. The web search tool incorporates the Serper search engine[5] and the Crawl4AI (UncleCode, 2024) parsing framework, and code execution is managed via the SandboxFusion environment (Bytedance-Seed-Foundation-Code-Team et al., 2025).

4.2 Main Results

Table 2 presents the comparative results of different frameworks, and the following conclusions are derived based on these results.

Capability-experience co-evolution achieves the strongest general agent. Mem²Evolve achieves

[4] https://openai.com/index/introducing-gpt-5/
[5] https://serpapi.com/

the best overall performance among all evaluated frameworks, demonstrating the effectiveness of jointly evolving capabilities and experience. Under the same GPT-5-chat as all baselines, Mem^{2}Evolve attains an average Pass@1 of 70.24% across all benchmarks, outperforming the strongest capability-centric baseline Alita by 6.46%, experience-centric baseline Aflow by 11.80%, and naive-llm by up to 18.53%. These consistent improvements confirm that capability–experience co-evolution yields a substantially more powerful general agent than either paradigm.

Breaking the Capability Boundaries of Static Agents. When initialized with only a Web Search tool, purely experience-centric methods yield marginal improvements over the base LLM. On GAIA, experience-centric baselines with a fixed toolset improve Pass@1 by at most 1.28%; on AIME, AFLOW achieves only +3.33% on AIME25 and no improvement on AIME24. In contrast, Mem^{2}Evolve, starting from the same minimal configuration but capable of evolving new tools and expert agents, achieves +57.84% on GAIA and +10.03%/+13.33% on AIME24/AIME25, respectively. These substantial gains demonstrate that Mem^{2}Evolve effectively extends the capability boundary of the base LLM.

Experience Memory Enhances Capability Expansion. Incorporating Experience Memory further enhances the effectiveness of capability expansion. Under matched conditions, Mem^{2}Evolve outperforms the capability-centric baseline Alita by 6.46% in average Pass@1. This improvement suggests that Experience Memory refines and stabilizes the utilization of newly evolved tools and agents, enabling capability expansion to translate more reliably into downstream performance gains.

5 Analysis

In this section, we conduct a comprehensive analysis to answer the following research questions **RQ1:** *What role does each module play in Mem^{2}Evolve?* (§5.1) **RQ2:** *How does experience guide asset generation?* (§5.2) **RQ3:** *How does Mem^{2}Evolve self-evolve in single task?* (§5.3) **RQ4:** *How does Mem^{2}Evolve self-evolve across tasks?* (§5.4) **RQ5:** *How does Mem^{2}Evolve behave in case studies?* (§D)

Framework	Avg. Pass@1	$\Delta^{\%}_{\text{rel}}$
Mem^{2}Evolve	**70.24**	–
w/o Asset Creation		
w/o Tool Creation	59.96	↓ 10.28
w/o Expert Agent Creation	68.52	↓ 1.72
w/o Experience Distillation		
w/o Tool Memory	67.11	↓ 3.13
w/o Agent Memory	65.51	↓ 4.73

Table 3: **Ablation study of Mem^{2}Evolve.** Full results are provided in Appendix 5.

5.1 RQ1: Ablation Study

To verify the effectiveness of each module in Mem^{2}Evolve, we conducted an ablation study on Asset Creation and Experience Distillation. As shown in Table 3, Mem^{2}Evolve consistently outperforms all variants, validating the necessity of the proposed Dual-Memory mechanism. Specifically, *w/o Tool Creation* causes the largest performance drop of 10.28%, highlighting that dynamically expanding the toolset is crucial for handling complex tasks, while *w/o Expert Agent Creation* still leads to a 1.72% decline because all tasks are forced onto a single general-purpose agent rather than expert agents. Moreover, removing *Agent Memory* causes a 4.73% performance drop, and removing *Tool Memory* causes a 3.13% performance drop, as this prevents the system from leveraging validated successes and past failures during both tool creation and task execution, making it difficult to reliably reproduce effective behaviors and avoid known mistakes, thereby degrading overall performance.

5.2 RQ2: Experience-Guided Asset Creation

In Section 5.1, we show that incorporating *Tool Memory* leads to consistent performance improvements. This section further investigates its impact on benchmarks that require extensive tool creation. We evaluate experience guidance using: (1) *First-Pass Validity*, which measures whether the initially generated tool satisfies the verification function $\text{Valid}(m_{\text{new}}, c_t)$ in Equation 8, and (2) *Avg. Improve Iter.*, defined as the average steps of $\text{Improve}(m_{\text{new}}, c_t)$ during the self-correction loop.

As shown in Table 4, experience guidance substantially improves the reliability and efficiency of tool creation. For AIME24/25, where the agent already demonstrates strong performance, experience guidance reduces the average number of debugging

Benchmark	w/o Exp.-Guide	w/ Exp.-Guide	$\Delta^{\%}_{\text{rel}}$
First-Pass Validity (↑)			
GAIA	32.7%	51.0% (+18.3%)	↑ 56.0
AIME24	64.9%	83.8% (+18.9%)	↑ 29.1
AIME25	61.8%	82.4% (+20.6%)	↑ 33.3
Avg.	53.1%	72.4% (+19.3%)	↑ 36.3
Avg. Improve Iter. (↓)			
GAIA	1.45	0.94 (-0.51)	↓ 35.2
AIME24	0.76	0.24 (-0.52)	↓ 68.4
AIME25	0.82	0.26 (-0.56)	↓ 68.3
Avg.	1.01	0.48 (-0.53)	↓ 52.5

Table 4: **Analysis of Experience-Guided Asset Creation.** We report first-pass validity and average improvement iterations on benchmarks requiring extensive tool generation. Experience guidance is associated with higher first-pass validity, with a relative improvement of up to 56.0% on GAIA, and fewer fix iterations, with reductions of nearly 68% on AIME benchmarks.

iterations by nearly 68% and increases first-pass validity to over 82%, indicating more accurate tool generation at the initial attempt. In contrast, in the more complex GAIA, experience guidance improves first-pass validity by 56.0% relative to the w/o Exp-Guide, suggesting that experience effectively constrains tool generation toward feasible solutions. Results and case in Figure 9 show that experience guidance significantly enhances the stability of tool generation, ensuring a more robust evolutionary trajectory for the agent.

5.3 RQ3: Single Task Self-Evolving

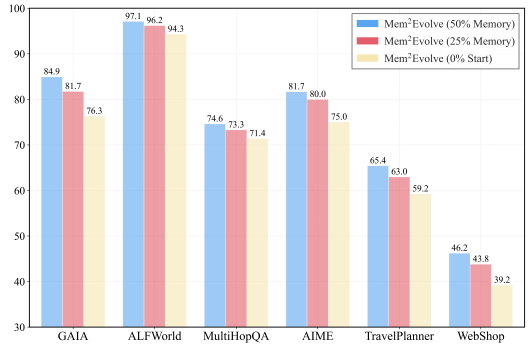


Figure 3: **Single-task self-evolving performance.** The results show that initializing the agent with prior memory consistently improves performance compared to the setting without initial memory, indicating that Mem^{2}Evolve can effectively leverage accumulated experience to enhance the task execution performance.

In Table 2, we evaluate the performance of Mem^{2}Evolve across multiple benchmarks, where each run starts without any pre-existing memory except for access to the web search tool. In this section, we further analyze the effect of introducing initial memory within the same task. Specifically, we construct initial memory using a subset of data from each benchmark and use it to initialize the system, after which evaluation is conducted on the remaining test set.

As shown in Figure 3, introducing initial memory consistently improves performance across all benchmarks compared to the setting without initial memory. Most of the performance gains are achieved with a relatively small amount of initial memory, while further enlarging the memory yields diminishing incremental improvements. This pattern suggests that memory accumulated within the same task is effective in enhancing agent performance, with early-stage memory capturing a large fraction of broadly applicable assets and high-utility experience.

5.4 RQ4: Cross Tasks Self-Evolving

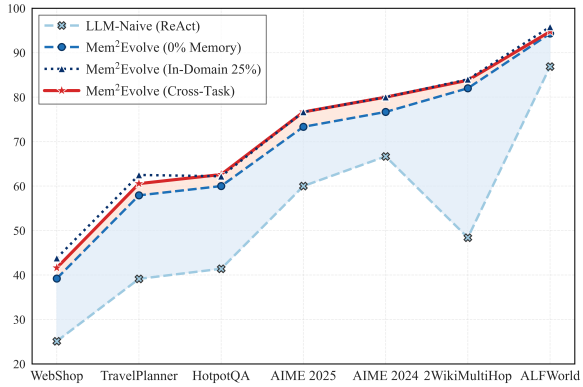


Figure 4: **Cross-task self-evolving performance.** When initialized with heterogeneous memory from GAIA, Mem^{2}Evolve consistently outperforms the setting without initial memory and achieves performance comparable to single-task initialization.

To evaluate the generalization capability of Mem^{2}Evolve in a cross-task setting, we initialize the agent with heterogeneous memory accumulated from GAIA and evaluate its performance across 7 target benchmarks.

As illustrated in Figure 4, cross-task memory initialization consistently improves performance compared to the setting without initial memory, and achieves results comparable to the 25% single-task initialization. Despite the domain mismatch between source and target tasks, the agent maintains stable evolutionary trajectories without suffering negative transfer. These results suggest that Mem^{2}Evolve can reuse heterogeneous memory

across tasks without adversely affecting performance. The structured representation of memory components and the retrieval mechanism contribute to this behavior by enabling selective access to task-relevant information.

6 Conclusion

We introduce Mem^{2}Evolve, a self-evolving agent framework that integrates Asset Memory and Experience Memory and enables their coordinated co-evolution. This design allows the agent to expand its capability space while continuously accumulating strategic experience, leading to more stable and sustained performance improvements. Extensive experimental results show that Mem^{2}Evolve consistently improves performance in both single-task and cross-task settings. We hope that Mem^{2}Evolve provides a practical foundation for building general-purpose, lifelong-learning agents with reduced reliance on human intervention.

Limitations

Mem^{2}Evolve is a self-evolving agent framework equipped with both asset memory and experience memory. During task execution, the framework dynamically generates expert agents and tools guided by past experience, thereby continuously expanding its capability boundaries while leveraging past experience to facilitate current task execution and achieve stable self-evolution. However, Mem^{2}Evolve relies on a sandbox environment to execute this autonomously generated code. This dependency limits the system's deployment scope, such as in open-world environments that require direct interaction with local file systems or unrestricted network access.

Acknowledgments

Thanks for the insightful comments and feedback from the reviewers. This work was supported by the National Natural Science Foundation of China (No. 62406015).

References

Sikai Bai, Haoxi Li, Jie Zhang, Yongjiang Liu, and Song Guo. 2026. Ttvs: Boosting self-exploring reinforcement learning via test-time variational synthesis. *Preprint*, arXiv:2604.08468.

Bytedance-Seed-Foundation-Code-Team, :, Yao Cheng, Jianfeng Chen, Jie Chen, Li Chen, Liyu Chen, Wentao Chen, Zhengyu Chen, Shijie Geng, Aoyan Li, Bo Li, Bowen Li, Linyi Li, Boyi Liu, Jiaheng Liu, Kaibo Liu, Qi Liu, Shukai Liu, and 37 others. 2025. Fullstack bench: Evaluating llms as full stack coders. *Preprint*, arXiv:2412.00535.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2024. Large language models as tool makers. In *The Twelfth International Conference on Learning Representations*.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Why do multi-agent llm systems fail? *Preprint*, arXiv:2503.13657.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje Karlsson, Jie Fu, and Yemin Shi. 2024. Autoagents: a framework for automatic agent generation. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pages 22–30.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, and 1 others. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*.

Zihao Cheng, Yuheng Lu, Huaiqian Ye, Zeming Liu, Minqi Wang, Jingjing Liu, Zihan Li, Wei Fan, Yuanfang Guo, Ruiji Fu, Shifeng She, Gang Wang, and Yunhong Wang. 2025a. Tcm-eval: An expert-level dynamic and extensible benchmark for traditional chinese medicine. *Preprint*, arXiv:2511.07148.

Zihao Cheng, Hongru Wang, Zeming Liu, Yuhang Guo, Yuanfang Guo, Yunhong Wang, and Haifeng Wang. 2025b. ToolSpectrum: Towards personalized tool utilization for large language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 20679–20699, Vienna, Austria. Association for Computational Linguistics.

Bin Deng, Yizhe Feng, Zeming Liu, Qing Wei, Xiangrong Zhu, Shuai Chen, Yuanfang Guo, and Yunhong Wang. 2025. RETAIL: Towards real-world travel planning for large language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 14881–14913, Suzhou, China. Association for Computational Linguistics.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, Zhaochun Ren, Nikos Aletras, Xi Wang, Han Zhou, and Zaiqiao Meng. 2025. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems. *Preprint*, arXiv:2508.07407.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin

Chi, and Wanjun Zhong. 2025. Retool: Reinforcement learning for strategic tool use in llms. *Preprint*, arXiv:2504.11536.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, and 1 others. 2025. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*.

GitHub. 2025. Spec kit: Toolkit to help you get started with spec-driven development. https://github.com/github/spec-kit. Accessed: 2025-12-19.

Ruiyan Han, Zhen Fang, XinYu Sun, Yuchen Ma, Ziheng Wang, Yu Zeng, Zehui Chen, Lin Chen, Wenxuan Huang, Wei-Jie Xu, and 1 others. 2026. Unicorn: Towards self-improving unified multimodal models through self-generated supervision. *arXiv preprint arXiv:2601.03193*.

Zhezheng Hao, Hong Wang, Jian Luo, Jianqing Zhang, Yuyan Zhou, Qiang Lin, Can Wang, Hande Dong, and Jiawei Chen. 2026. Recreate: Reasoning and creating domain agents driven by experience. *arXiv preprint arXiv:2601.11100*.

Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. 2020. Constructing a multi-hop QA dataset for comprehensive evaluation of reasoning steps. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 6609–6625, Barcelona, Spain (Online). International Committee on Computational Linguistics.

Ruili Jiang, Kehai Chen, Xuefeng Bai, Zhixuan He, Juntao Li, Muyun Yang, Tiejun Zhao, Liqiang Nie, and Min Zhang. 2025. A survey on human preference learning for aligning large language models. *ACM Computing Surveys*, 58(6):1–39.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, and 1 others. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*.

Dawei Li, Bohan Jiang, Liangjie Huang, Alimohammad Beigi, Chengshuai Zhao, Zhen Tan, Amrita Bhattacharjee, Yuxuan Jiang, Canyu Chen, Tianhao Wu, and 1 others. 2025a. From generation to judgment: Opportunities and challenges of llm-as-a-judge. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 2757–2791.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008.

Xiaoxi Li, Wenxiang Jiao, Jiarui Jin, Guanting Dong, Jiajie Jin, Yinuo Wang, Hao Wang, Yutao Zhu, Ji-Rong Wen, Yuan Lu, and Zhicheng Dou. 2025b. Deepagent: A general reasoning agent with scalable toolsets. *Preprint*, arXiv:2510.21618.

Jingjing Liu, Zeming Liu, Zihao Cheng, Mengliang He, Xiaoming Shi, Yuhang Guo, Xiangrong Zhu, Yuanfang Guo, Yunhong Wang, and Haifeng Wang. 2025a. RepoDebug: Repository-level multi-task and multi-language debugging evaluation of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 23784–23813, Suzhou, China. Association for Computational Linguistics.

Zeming Liu, Haifeng Wang, Zheng-Yu Niu, Hua Wu, Wanxiang Che, and Ting Liu. 2020. Towards conversational recommendation over multi-type dialogs. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 1036–1049.

Zhipeng Liu, Xuefeng Bai, Kehai Chen, Xinyang Chen, Xiucheng Li, Yang Xiang, Jin Liu, Hong-Dong Li, Yaowei Wang, Liqiang Nie, and 1 others. 2025b. A survey on the feedback mechanism of llm-based ai agents. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pages 10582–10592. International Joint Conferences on Artificial Intelligence.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. 2023. Dynamic llm-agent network: An llm-agent collaboration framework with agent team optimization. *arXiv preprint arXiv:2310.02170*.

Yunbo Long, Yuhan Liu, and Liming Xu. 2026. Emomas: Emotion-aware multi-agent system for high-stakes edge-deployable negotiation with bayesian orchestration. *Preprint*, arXiv:2604.07003.

Kexin Ma, Bojun Li, Yuhua Tang, Liting Sun, and Ruochun Jin. 2026. Cast: Character-and-scene episodic memory for agents. *Preprint*, arXiv:2602.06051.

Xiaowen Ma, Chenyang Lin, Yao Zhang, Volker Tresp, and Yunpu Ma. 2025. Agentic neural networks: Self-evolving multi-agent systems via textual backpropagation. *arXiv preprint arXiv:2506.09046*.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2024. GAIA: a benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations*.

OpenAI. 2025. Introducing deep research. https://openai.com/index/introducing-deep-research/. Accessed: 2026-01-02.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T Le, Samira Daruki, Xiangru Tang, and 1 others. 2025. Reasoningbank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140*.

Jean Piaget. 1972. Development and learning. *Reading in child behavior and development*, pages 38–46.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, and 1 others. 2025. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*.

Deng Qiyuan, Kehai Chen, Min Zhang, and Zhongwen Xu. 2026. HiPO: Self-hint policy optimization for RLVR. In *The Fourteenth International Conference on Learning Representations*.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. 2025. Agentsquare: Automatic LLM agent search in modular design space. In *The Thirteenth International Conference on Learning Representations*.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2020. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*.

Linxin Song, Jiale Liu, Jieyu Zhang, Shaokun Zhang, Ao Luo, Shijian Wang, Qingyun Wu, and Chi Wang. 2024. Adaptive in-conversation team building for language model agents. *arXiv preprint arXiv:2405.19425*.

UncleCode. 2024. Crawl4ai: Open-source llm friendly web crawler & scraper. https://github.com/unclecode/crawl4ai.

Guancheng Wan, Mo Zhou, Ziyi Wang, Xiaoran Shang, Eric Hanchen Jiang, Guibin Zhang, Jinhe Bi, Yunpu Ma, Zaixi Zhang, Ke Liang, and 1 others. 2026. Dawn: Distributed llm multi-agent workflow synthesis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 26099–26106.

Hongru Wang, Cheng Qian, Manling Li, Jiahao Qiu, Boyang Xue, Mengdi Wang, Heng Ji, and Kam-Fai Wong. 2025a. Toward a theory of agents as tool-use decision-makers. *arXiv preprint arXiv:2506.00886*.

Hongru Wang, Cheng Qian, Wanjun Zhong, Xiusi Chen, Jiahao Qiu, Shijue Huang, Bowen Jin, Mengdi Wang, Kam-Fai Wong, and Heng Ji. 2025b. Acting less is reasoning more! teaching model to act efficiently. *arXiv preprint arXiv:2504.14870*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Georg Wölflein, Dyke Ferber, Daniel Truhn, Ognjen Arandjelovic, and Jakob Nikolas Kather. 2025. LLM agents making agent tools. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26092–26130, Vienna, Austria. Association for Computational Linguistics.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. 2024. Travelplanner: a benchmark for real-world planning with language agents. In *Proceedings of the 41st International Conference on Machine Learning*, pages 54590–54613.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, Brussels, Belgium. Association for Computational Linguistics.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022a. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757.

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

yt-dlp. 2025. yt-dlp: A feature-rich command-line audio/video downloader. https://github.com/yt-dlp/yt-dlp. Accessed: 2025-12-19.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Dongsheng Li, and Deqing Yang. 2025. Evoagent: Towards automatic multi-agent generation via evolutionary algorithms. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 6192–6217.

Liang Yue, Yihong Tang, Kehai Chen, Jie Liu, and Min Zhang. 2025. Master: Enhancing large language model via multi-agent simulated teaching. *arXiv preprint arXiv:2506.02689*.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. 2025. Optimizing generative ai by backpropagating language model feedback. *Nature*, 639(8055):609–616.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, and 1 others. 2025a. Aflow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*.

Wenyuan Zhang, Xinghua Zhang, Haiyang Yu, Shuaiyi Nie, Bingli Wu, Juwei Yue, Tingwen Liu, and Yongbin Li. 2026. Expseek: Self-triggered experience seeking for web agents. *Preprint*, arXiv:2601.08605.

Yao Zhang, Chenyang Lin, Shijie Tang, Haokun Chen, Shijie Zhou, Yunpu Ma, and Volker Tresp. 2025b. Swarmagentic: Towards fully automated agentic system generation via swarm intelligence. *arXiv preprint arXiv:2506.15672*.

Hongli Zhou, Hui Huang, Ziqing Zhao, Lvyuan Han, Huicheng Wang, Kehai Chen, Muyun Yang, Wei Bao, Jian Dong, Bing Xu, and 1 others. 2026. Lost in benchmarks? rethinking large language model benchmarking with item response theory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 35085–35093.

A Mem^{2}Evolve

A.1 Defining Continuous and Stable Evolving

Back to ToC

We define two fundamental characteristics of self-evolving agents: **(1)** the ability to continuously evolve with minimal human intervention by persistently expanding their capabilities to solve unseen tasks, and **(2)** the capability to efficiently leverage past experience, enabling correct experience transfer and effective error avoidance for seen tasks, either offline or online.

- **Optimization.** An agent system should be capable of automatically optimizing its internal instructions and coordination workflows based on environmental feedback, thereby achieving optimal task-specific performance. Traditional agent development paradigms rely heavily on manual prompt engineering or predefined interaction protocols. Such approaches are not only labor-intensive but also poorly suited to dynamically evolving task requirements. Therefore, an effective self-evolution framework should emulate a form of "backpropagation" mechanism, continuously refining agent role definitions, prompting strategies, and even multi-agent collaboration topologies through feedback signals or textual gradients derived from task execution outcomes. Crucially, this optimization should not be limited to transient runtime adjustments but should fundamentally enhance the system's intrinsic competence in handling similar tasks.

- **Experience Persistence.** To enable genuine lifelong learning, the framework must be able to transform both successful strategies and failure cases from historical tasks into long-term memory assets that persist beyond the lifecycle of a single task. Many existing methods reset system states after task completion, forcing agents to re-explore from scratch when encountering similar scenarios. This not only wastes computational resources but also allows recurring errors. Hence, a cross-task experience persistence mechanism is essential. Whether implemented via explicit external databases that store reasoning trajectories or via implicit knowledge internalization through parameter or prompt updates, this mechanism should enable rapid retrieval and reuse of prior knowledge when facing new tasks, thereby mitigating the cold-start problem and avoiding known pitfalls.

- **Agent Creation.** The framework should not depend on predefined expert agent modules with fixed roles, prompts, or decision logics. Instead, it should be capable of dynamically constructing optimal expert agent teams conditioned on task demands. This capability is critical for high-level, complex planning tasks, where increasing task complexity typically entails decomposing the problem into multiple subtasks with distinct objectives. A single general-purpose agent is often insufficient to handle all subtasks effectively, necessitating specialized agents that each address the components they are best suited for. However, given the vast diversity of real-world tasks, manually predefined expert agents cannot cover all possible scenarios. The system must therefore support dynamic, task-driven agent generation.

- **Tool Creation.** By invoking tools, agent systems can substantially expand their capability boundaries and overcome the limitations imposed by static knowledge. Tools enable access to real-time information, execution of complex mathematical reasoning, and completion of specialized operations. However, task-specific tools typically require careful manual design. When confronted with general, previously unseen tasks, human developers are often still required to create new tools, which is inherently unscalable. To enable continual capability expansion, the framework must therefore possess the ability to autonomously generate tools.

- **Experience-Guided Creation.** When encountering unseen tasks that require the generation of new agents or tools, the framework should leverage its internal assets and accumulated memory to guide the creation process. For example, when a new task involves parsing YouTube video subtitles, previously generated tool documentation for downloading YouTube videos can serve as a reference to facilitate new tool construction. This experience-guided mechanism improves the stability of generated tools and agents, reduces randomness and hallucinations in large lan-

guage model outputs, and thereby enables a more robust and reliable evolutionary process.

A.2 Evaluation of Existing Self-Evolving Agent Frameworks

Back to ToC

- **DyLAN** (Liu et al., 2023) models multi-agent collaboration as a Temporal Feed-Forward Network, implementing a "Team Optimization" stage that utilizes a backward message-passing mechanism to calculate "Agent Importance Scores" based on unsupervised peer ratings. **DyLAN satisfies Optimization:** it actively employs environmental feedback to refine the collaboration topology iteratively. By identifying and selecting the most contributory agents while pruning low-performing ones, DyLAN automatically optimizes the team composition and interaction structure for specific tasks, aligning with the definition of optimizing collaboration logic and topology. However, **DyLAN fails Agent Creation and Tool Creation:** it does not generate new expert definitions or tools from scratch; instead, it relies on selecting the best subset from a fixed, pre-defined candidate pool of agents. Finally, it **offers limited Experience Persistence:** while it can reuse calculated importance scores for similar tasks, it lacks a semantic memory mechanism to guide the generation of new assets for entirely unseen domains.

- **DSPy** (Khattab et al., 2023) introduces a programming model that abstracts language model pipelines as text transformation graphs, allowing users to define declarative modules (e.g., ChainOfThought) via natural language signatures. **DSPy satisfies Optimization**: it employs a compiler with various "teleprompters" to automatically refine the pipeline's instructions or fine-tune the underlying language model weights based on metric-driven feedback and bootstrapped demonstrations. However, **DSPy fails Agent Creation and Tool Creation**: it relies on the user to explicitly define the program structure, the flow of control, and the specific modules and tools to be used, rather than autonomously synthesizing new agent roles or executable tools from scratch. Furthermore, **DSPy fails Experience Persistence**: it treats experience utilization as a discrete compilation process rather than a continuous memory accumulation. Once the pipeline is compiled, the historical traces are frozen into static few-shot examples or weights, lacking a dynamic, retrievable memory bank to persistently store and reuse new inference trajectories for future tasks.

- **ReasoningBank** (Ouyang et al., 2025) introduces a memory framework that distills generalizable reasoning strategies from both successful and failed trajectories, enabling agents to retrieve relevant insights for new tasks. **ReasoningBank satisfies Experience Persistence**: it explicitly stores abstracted reasoning patterns in a long-term memory bank, allowing the system to mitigate the cold-start problem by transferring knowledge across tasks and preventing the repetition of past errors. However, **ReasoningBank fails Optimization**: while it improves performance via RAG, it does not structurally optimize the agent's topology, internal prompt templates, or parameters. The agent's core functionality remains static, relying on external memory injection rather than internal refinement. It likewise fails **Agent Creation** and **Tool Creation**, as it operates with a fixed agent architecture (e.g., ReAct), leveraging dynamic memory, rather than generating new agent entities or executable tools from scratch. Consequently, it also fails **Experience-Guided Creation**, as there is no asset generation process to be guided by its rich memory.

- **AFlow** (Zhang et al., 2025a) reformulates agentic system development as a search problem over code-represented workflows, utilizing Monte Carlo Tree Search (MCTS) to iteratively explore and optimize the design space. **AFlow satisfies Optimization**: it treats the workflow structure and node-level prompts as hyperparameters, optimizing them based on execution feedback (e.g., success rates, costs) to find the most effective graph topology. It also satisfies **Agent Creation** and **Experience-Guided Creation**: the framework autonomously constructs new workflow nodes (effectively new agents) and connections by leveraging the search history (MCTS values) to guide the generation process, moving away from manual engineering. However,

AFlow fails Tool Creation: it focuses on orchestrating the flow of LLM calls and existing tools rather than synthesizing new executable tool code from scratch. Furthermore, it fails **Experience Persistence**: the experience is utilized only during the offline search phase to produce a static, compiled workflow; it lacks a dynamic, long-term memory mechanism to continuously accumulate and retrieve reasoning patterns for lifelong learning across different tasks.

- **AgentSquare** (Shang et al., 2025) proposes a search-based framework that automates the design of LLM agents by exploring a modular space comprising planning, reasoning, tool use, and memory modules. **AgentSquare satisfies Optimization**: it treats the agent design process as an objective function maximization problem, iteratively refining the agent's architecture (i.e., module combinations) based on evaluation feedback to find the optimal configuration. It also satisfies **Agent Creation** and **Experience-Guided Creation**: the framework autonomously synthesizes new agent instances by recombining modules and utilizes an "Experience Pool" (containing history of evaluated agent-task pairs) to train a surrogate model, which efficiently guides the generation of new candidates towards high-performance regions. It further satisfies **Experience Persistence** by explicitly storing these search trajectories and evaluation results in the Experience Pool, enabling the system to learn from past search iterations. However, **AgentSquare fails Tool Creation**: while it optimizes the *mechanism* of tool usage (e.g., choosing between ReAct or Plan-and-Solve), it relies on orchestrating existing tools rather than generating new executable tool implementations from scratch to extend capabilities.

- **ANN** (Ma et al., 2025) conceptualizes multi-agent systems as neural networks, treating agents as learnable nodes and their communication as edges. **ANN satisfies Optimization**: drawing inspiration from backpropagation, it introduces a "textual backpropagation" mechanism that computes textual gradients based on error feedback to iteratively update the agents' system prompts (which function as learnable weights). However, **ANN fails Agent Creation and Tool Creation**: the framework operates on pre-defined architectural topologies (e.g., Chain, Stack, or Grid structures) with a fixed number of agent nodes; it optimizes the *behavior* of these existing agents rather than autonomously synthesizing new agent roles or executable tools from scratch. Furthermore, it fails **Experience Persistence**: similar to traditional model training, the learned experience is implicitly internalized into the optimized prompt parameters during the optimization phase. It lacks a dynamic, explicitly retrievable memory bank to persistently store reasoning trajectories, limiting its ability to support lifelong learning or transfer insights to entirely new domains without re-training. Consequently, it also fails **Experience-Guided Creation**.

- **Alita** (Qiu et al., 2025) introduces a self-evolving framework that enables an LLM agent to dynamically expand its capability boundaries by creating and integrating new tools. **Alita satisfies Tool Creation**: it employs a "Creator" module that autonomously synthesizes executable Python tools from scratch to address tasks where existing tools are insufficient. It also satisfies **Optimization** and **Experience Persistence**: the framework maintains an explicit "Experience Pool" of successful tool-use trajectories and utilizes a "Promptist" module to retrieve relevant demonstrations and refine the agent's prompts based on execution feedback. However, **Alita fails Agent Creation**: it operates as a single-agent system that evolves its tool library, rather than synthesizing new agent roles or collaborative teams. Furthermore, it fails **Experience-Guided Creation** (in the context of asset generation): while it uses experience to optimize *usage* prompts, the generation of *new tools* is driven by immediate task failures and reflection, without leveraging a retrieval mechanism over historical creation artifacts to guide the synthesis process.

- **ToolMaker** (Wölflein et al., 2025) introduces a dual-LLM framework where a "Tool Maker" autonomously generates reusable Python tools (functions) to address specific tasks, which are then utilized by a "Tool User" for subsequent problem-solving. **ToolMaker satisfies Tool**

Creation: its core mechanism is the synthesis of executable code to encapsulate reasoning steps into reusable tools, thereby explicitly expanding the agent's action space. It also satisfies **Optimization**: during the tool creation phase, it employs a verification loop (utilizing unit tests) to validate the generated code and uses error feedback to iteratively refine and debug the tool until it functions correctly. However, **ToolMaker fails Agent Creation**: the framework operates with fixed, pre-defined roles (Maker and User) rather than synthesizing new agent personas or collaborative team structures from scratch. Furthermore, it fails **Experience Persistence** (in the context of cross-task learning) and **Experience-Guided Creation**: while the generated tools are stored for reuse on instances of the *same* task, the framework does not maintain a retrievable memory bank of creation strategies or past artifacts to guide the generation process for entirely *new, unseen* tasks, effectively facing the cold-start problem for each new domain.

- **AgentVerse** (Chen et al., 2023) proposes a flexible multi-agent framework that orchestrates the problem-solving process through four key stages: Expert Recruitment, Collaborative Decision Making, Action Execution, and Evaluation. **AgentVerse satisfies Agent Creation**: utilizing the Expert Recruitment mechanism, the framework autonomously generates and customizes new agent roles and descriptions tailored to the specific progress of the task, rather than relying on a fixed set of pre-defined personas. It also satisfies **Optimization**: the Evaluation stage provides real-time feedback on the agents' outcomes, which is used to iteratively refine the collaborative decision-making process and adjust the team's composition or strategies during runtime. However, **AgentVerse fails Tool Creation**: while agents can utilize existing tools or execute code, the framework focuses on evolving the *team structure* and *agent personas* rather than synthesizing new, reusable executable tool definitions from scratch to expand the action space. Furthermore, it fails **Experience Persistence** and **Experience-Guided Creation**: the optimization is confined to the immediate context of the current task loop; it lacks a long-term, retrievable memory mechanism to store successful collaboration patterns or reasoning trajectories for future cross-task transfer, meaning each new task session effectively starts without historical guidance.

- **AutoAgents** (Chen et al., 2024) introduces an automatic agent generation framework that dynamically synthesizes a team of specialized agents (including roles and expert profiles) tailored to the specific input task. **AutoAgents satisfies Agent Creation**: it leverages a "Planner" to decompose the task and autonomously generate the identities and descriptions of the necessary expert agents, rather than selecting from a pre-defined library. It also satisfies **Optimization**: it employs an "Observer" mechanism (Agent Observer and Plan Observer) to review the generated agents and execution plans, providing feedback to refine and optimize the team structure and workflows before execution. However, **AutoAgents fails Tool Creation**: while the generated agents can utilize existing tools, the framework focuses on synthesizing the *agents* themselves, not generating new executable tool code from scratch to expand the system's capabilities. Furthermore, it fails **Experience Persistence** and **Experience-Guided Creation**: the generation process is effectively zero-shot for each new task instance; it does not maintain a long-term, retrievable memory of past successful agent configurations or planning trajectories to guide the generation of future agents, tackling each task as an isolated event without accumulation of experience.

- **SwarmAgentic** (Zhang et al., 2025b) applies Swarm Intelligence (SI) principles (specifically Particle Swarm Optimization) to the domain of agent system design, treating agents and tools as particles that evolve in a search space. **SwarmAgentic satisfies Agent Creation and Tool Creation**: the framework autonomously synthesizes both the agent definitions (roles, prompts) and executable tool code (Python functions) from scratch to construct a functional system, rather than selecting from a fixed library. It also satisfies **Optimization**: it utilizes a velocity-based update mechanism to iteratively refine the agents'

prompts and tools based on the "personal best" and "global best" feedback found during the swarm search process. However, **SwarmAgentic fails Experience Persistence**: the optimization history and learned patterns are transient, utilized only to converge on a solution for the current task instance. It does not maintain a persistent, retrievable memory bank of successful designs to support lifelong learning across different tasks. Consequently, it also fails **Experience-Guided Creation**, as the initialization of new systems for unseen tasks relies on zero-shot generation rather than being informed by a repository of historical assets.

A.3 Task Planning

Back to ToC

In real-world settings, tasks are typically accomplished through multiple steps. To improve the specialization of subsequently created tools and to fully leverage expert agents by assigning them distinct roles aligned with their respective strengths, the system first decomposes a complex task into a set of sub-tasks during the planning stage. Each sub-task specifies its objective, the expected output format, and its dependencies on other sub-tasks—namely, the results from prerequisite sub-tasks required for execution.

A.4 Tool Creation

Back to ToC

When a sub-task exceeds the agent's existing capability scope, the system initiates a tool-creation workflow to expand its capability boundaries. However, synthesizing effective tools based solely on sub-task descriptions proves inadequate. Our empirical analysis identifies three key limitations: (1) brief sub-task descriptions provide insufficient constraints, resulting in unstable and inconsistent tool generation; (2) tools derived solely from the model's internal, static knowledge often lack practical usability; and (3) the inherent stochasticity of model outputs hinders the reproducibility of successful tool-generation processes and prevents effective reuse of failure cases.

To overcome these challenges, we introduce a three-stage tool synthesis strategy: (1) tool specification generation to formalize functionality and interfaces; (2) tool documentation and experience collection to ground tool usage and accumulate actionable knowledge; and (3) tool implementation to produce reliable and reusable tools.

- **Tool Spec Generation.** Inspired by specification-driven development paradigms (GitHub, 2025; Zhang et al., 2026), we require the model to first generate a formal tool specification before implementing the tool itself. As illustrated in Figure 5, this specification includes the *tool name*, a *concise description*, *input parameters*, *output format*, and *core logic*. The core logic is articulated as a sequence of concrete steps that explicitly describe the tool's execution process (e.g., "Step 1: validate input compliance"). By introducing this intermediate specification stage, the agent can generate tools by directly adhering to well-defined requirements, thereby improving controllability, consistency, and overall accuracy in the tool creation process.

- **Tool Documentation and Experience Collection.** To further mitigate the limitation of relying solely on the parametric, static knowledge, we incorporate an additional grounding step prior to tool generation. Specifically, the agent leverages a web search tool to retrieve external documentation, such as open-source tool descriptions on GitHub[6] and debugging discussions from Stack Overflow[7]. In parallel, the agent queries its Experience Memory using the generated tool specification to retrieve relevant development references. For example, both a tool for extracting basic YouTube video metadata and a tool for downloading YouTube videos can be grounded in documentation for the open-source utility *yt-dlp* (yt-dlp, 2025). By integrating externally sourced documents with experience-based retrieval, the tool creation process is better grounded in real-world implementations, leading to more practical and reliable tools.

- **Tool Implementation.** With a well-defined tool specification and sufficiently rich tool documentation and implementation experience stored in memory, the system can proceed to generate the corresponding tool. As shown in Code 1, we intentionally encapsulate each tool in an MCP-compliant format, ensuring that it can be seamlessly integrated by different LLM backbones. This design enables

[6] https://github.com
[7] https://stackoverflow.com/

```
[
    {
        "tool_name": "simulate_piston_platform_game",
        "tool_description": "Simulates a specific ping-pong game mechanism involving a ball queue, a
    limited-capacity platform, and three random pistons with complex ejection/replacement rules.
    Used to calculate the probability of each ball number being 'ejected by a piston' (winning)
    versus being 'released' (eliminated).",
        "input_parameters": [
            {
                "name": "num_balls",
                "type": "integer",
                "description": "Total number of balls in the queue, numbered 1 to N.",
                "default": 100
            },
            {
                "name": "num_simulations",
                "type": "integer",
                "description": "Monte Carlo iterations.",
                "default": 100000
            }
        ],
        "output_format": {
            "type": "object",
            "properties": {
                "win_probabilities": {
                    "type": "object",
                    "description": "Mapping of ball number to its probability of being ejected by a
    piston (Winning)."
                },
                "best_choice": {
                    "type": "integer",
                    "description": "The ball number with the highest win probability."
                }
            }
        },
        "core_logic": [
            "Step 1: Initialize `win_counts` for all balls to 0.",
            "Step 2: Run loop `num_simulations` times.",
            "Step 3: In each sim, initialize a queue `deck` [1..num_balls] and a `platform` holding
    the first 3 balls.",
            "Step 4: While platform is not empty, randomly select a piston (1, 2, or 3) with equal
    probability.",
            "Step 5: Apply Piston Rules:",
            "   - If Piston 1: Eject Pos 1 (Win). Move Pos 2->1, Pos 3->2. Refill 1 ball from deck to
     Pos 3.",
            "   - If Piston 2: Eject Pos 2 (Win). Release Pos 1 (Loss/Die). Move Pos 3->1. Refill 2
    balls from deck to Pos 2 & 3.",
            "   - If Piston 3: Eject Pos 3 (Win). Release Pos 1 (Loss/Die). Move Pos 2->1. Refill 2
    balls from deck to Pos 2 & 3.",
            "Step 6: If a ball is 'Ejected' (Win), increment its count in `win_counts`. If 'Released'
    , do nothing.",
            "Step 7: Continue until platform and deck are empty.",
            "Step 8: Calculate probabilities = wins / total_simulations."
        ]
    }
]
```

Figure 5: **Specification of the tool for simulating the Piston Platform game.** The specification includes the tool name and description, detailed definitions of input parameters and output formats—where each parameter is characterized by its name, type, description, and default value—as well as the core logic of the tool implementation, guiding subsequent tool creation.

model-agnostic interoperability and facilitates efficient reuse in subsequent applications.

A.5 Assets Recruitment

Back to ToC

To optimize the utilization of tools and expert agents stored in Asset Memory, Mem^{2}Evolve implements an *Assets Recruitment* phase prior to task execution. This mechanism operates at the granularity of sub-tasks. Let $\mathbf{q} = embedding(s_i)$ denote the embedding of the current sub-task description.

Expert Agent Retrieval We query the agent memory $\mathcal{M}_{agt}$ to find the most proficient expert. The retrieval key for a candidate agent a_i is defined as $\mathbf{h}_{a_i} = embedding(\rho_i \oplus \epsilon_i \oplus \sigma_i)$, derived from its role, expertise, and behavior suggestions. We select the optimal agent a^* by retrieving the Top-1 candidate that exceeds a similarity threshold δ:

$$a^* = \arg\max_{a_i \in \mathcal{M}_{agt}} \Big\{ \cos(\mathbf{q}, \mathbf{h}_{a_i}) \mid \cos(\mathbf{q}, \mathbf{h}_{a_i}) > \delta \Big\} \tag{12}$$

If the set is empty, a new agent generation process is triggered.

Tool Retrieval Similarly, for tool memory $\mathcal{M}_{tool}$, the key is $\mathbf{h}_{t_j} = embedding(n_j \oplus d_{func,j})$. To balance functional support with context window constraints, we retrieve the Top-K relevant tools to form the available toolset $\mathbb{T}_{avail}$:

$$\mathbb{T}_{avail} = \underset{t_j \in \mathcal{M}_{tool}}{\text{Top-K}} \Big\{ t_j \mid \cos(\mathbf{q}, \mathbf{h}_{t_j}) > \delta \Big\} \tag{13}$$

A.6 Execution

Back to ToC

Since expert agents must frequently invoke tools to interact with the external environment and make subsequent decisions based on environmental feedback, we adopt the ReAct (Yao et al., 2022b) framework, which alternates among **Think**, **Action**, and **Observation** steps. Specifically, we design a standardized ReAct system prompt template with dedicated placeholders for prerequisite results from dependent sub-tasks, the expert role, task-specific suggestions, and the set of available tools. During execution, these placeholders are dynamically populated with agent-specific content for each expert agent. This templated design ensures both the stability of the reasoning–action loop and the extensibility of the framework across different expert roles and task settings.

B Experimental Details

B.1 Baselines

Back to ToC

In this section, we provide detailed descriptions of the baseline frameworks employed in our evaluation, categorized by their operational paradigms.

Naive Large Language Models

- **Direct**: This is the most fundamental approach, where the task description is fed directly into the Large Language Model (LLM) without any intermediate reasoning steps or external tools. It serves as a baseline to measure the inherent zero-shot capability of the model.
- **CoT** (Wei et al., 2022): CoT enhances the reasoning capabilities of LLMs by prompting them to generate a series of intermediate reasoning steps before producing the final answer. This method is particularly effective for complex tasks requiring multi-step logic.
- **ReAct** (Yao et al., 2022b): ReAct synergizes reasoning and acting by allowing the model to generate reasoning traces and task-specific actions (such as web searches) in an interleaved manner. This enables the agent to interact with external environments to retrieve information and update its context dynamically.
- **OpenAI Deep Research**: A commercial-grade autonomous research agent developed by OpenAI. It is designed to perform deep, multi-step research tasks by browsing the web, synthesizing information from multiple sources, and generating comprehensive reports, representing the state-of-the-art in proprietary agent systems.

Experience-Centric Frameworks

- **DyLAN** (Liu et al., 2023): DyLAN is a framework that models multi-agent collaboration as a Temporal Feed-Forward Network. It introduces a "Team Optimization" stage utilizing a backward message-passing mechanism to calculate Agent Importance Scores. By actively identifying high-contribution agents and pruning low-performing ones, DyLAN iteratively optimizes the team's collaboration topology, though it relies on a fixed pool of agents rather than creating new ones.

```
{
        "role": "Probability Simulation Analyst",
        "expertise": "Specializes in stochastic modeling and quantitative analysis to derive
    probabilities from complex mechanical simulations.",
        "suggestions": [
                "Execute multiple simulation runs to ensure statistical significance of the results
    .",
                "Aggregate ejection data to calculate the specific probability for each ball.",
                "Identify the ball with the maximum ejection frequency from the dataset."
        ],
        "tools": [
                "simulate_ping_pong_game"
        ]
}
```

Figure 6: **Specification of the probabilistic simulation expert.** The specification defines the expert agent's role, areas of expertise, suggested strategies or recommendations, and the list of tools available for use during task execution.

Method	GAIA	Embodied	Multi-Hop QA		Math		Planning	Web	Avg.
	Total	ALFWorld	HotpotQA	2Wiki	AIME24	AIME25	TravelPlanner	WebShop	
Mem²Evolve (Ours)	**76.31**	**94.31**	**60.80**	**82.00**	**76.70**	**73.33**	**59.25**	**39.20**	**70.24**
w/o Asset Creation									
w/o Tool Creation	21.69 (↓ 54.62)	94.10 (↓ 0.21)	59.50 (↓ 1.30)	81.50 (↓ 0.50)	66.67 (↓ 10.03)	60.00 (↓ 13.33)	57.15 (↓ 2.10)	39.05 (↓ 0.15)	59.96 (↓ 10.28)
w/o Expert Agent Creation	75.30 (↓ 1.01)	93.65 (↓ 0.66)	59.00 (↓ 1.80)	81.00 (↓ 1.00)	73.33 (↓ 3.37)	70.00 (↓ 3.33)	57.08 (↓ 2.17)	38.80 (↓ 0.40)	68.52 (↓ 1.72)
w/o Experience Distillation									
w/o Tool Memory	67.47 (↓ 8.84)	94.25 (↓ 0.06)	60.00 (↓ 0.80)	81.50 (↓ 0.50)	70.00 (↓ 6.70)	66.67 (↓ 6.66)	57.85 (↓ 1.40)	39.15 (↓ 0.05)	67.11 (↓ 3.13)
w/o Agent Memory	74.70 (↓ 1.61)	88.06 (↓ 6.25)	56.80 (↓ 4.00)	77.40 (↓ 4.60)	73.33 (↓ 3.37)	70.00 (↓ 3.33)	49.36 (↓ 9.89)	34.40 (↓ 4.80)	65.51 (↓ 4.73)

Table 5: **Ablation study of Mem²Evolve**. Pass@1 scores are reported. Performance drops (↓) relative to the full model are shown in parentheses.

- **EvoAgent** (Yuan et al., 2025): EvoAgent applies evolutionary algorithms to multi-agent systems, treating agents as individuals in a population. It employs operations such as crossover and mutation on agent prompts to iteratively evolve their behaviors. This allows the system to discover more effective agent personas and strategies over time without manual prompt engineering.

- **AFlow** (Zhang et al., 2025a): AFlow reformulates agentic system development as a search problem over code-represented workflows. Utilizing Monte Carlo Tree Search (MCTS), it iteratively explores and optimizes the design space of workflow structures and node-level prompts. AFlow autonomously constructs new workflow nodes and connections guided by search history, moving away from manual engineering to find the most effective graph topology for specific tasks.

- **DSPy** (Khattab et al., 2023): DSPy introduces a programming model that abstracts LM pipelines as text transformation graphs. It employs a compiler with various "teleprompters" to automatically refine the pipeline's instructions or fine-tune the underlying language model weights based on metric-driven feedback. While it optimizes the pipeline effectively, it relies on user-defined program structures rather than autonomously synthesizing new agent roles or tools.

Capability-Centric Frameworks (Tool-Generative)

- **Alita**[8] (Qiu et al., 2025): Alita is a self-evolving framework designed to dynamically expand an agent's capability boundaries. It features a "Creator" module that autonomously synthesizes executable Python tools from scratch to address tasks where existing tools are insufficient. Additionally, Alita utilizes a "Promptist" module and an explicit Experience Pool to refine usage prompts based on execution feedback, enabling continuous adaptation.

- **ToolMaker** (Wölflein et al., 2025): ToolMaker adopts a dual-LLM framework comprising a "Tool Maker" and a "Tool User." The Maker autonomously generates reusable Python tools (functions) to encapsulate reasoning steps for specific tasks, while the User applies them for problem-solving. It includes a verification loop with unit tests to ensure the reliability of the generated code, explicitly expanding the agent's action space through code synthesis.

Capability-Centric Frameworks (Agent-Generative)

- **AgentVerse** (Chen et al., 2023): AgentVerse proposes a flexible multi-agent framework orchestrating the problem-solving process through Expert Recruitment, Collaborative Decision Making, Action Execution, and Evaluation. It autonomously generates and customizes new agent roles tailored to the task progress. The framework uses real-time feedback to refine the collaborative process and adjust team composition during runtime.

- **AutoAgents** (Chen et al., 2024): AutoAgents introduces an automatic agent generation framework that dynamically synthesizes a team of specialized agents tailored to the input task. Leveraging a "Planner" to decompose tasks, it autonomously generates expert identities and descriptions. An "Observer" mechanism further reviews and refines the execution plans and team structure, ensuring the generated agents are optimized for the specific problem context.

- **SwarmAgentic** (Zhang et al., 2025b): SwarmAgentic applies Swarm Intelligence principles, specifically Particle Swarm Optimization (PSO), to agent system design. It treats agents and tools as particles evolving in a search space, autonomously synthesizing both agent definitions and executable tool code. The framework utilizes velocity-based updates to iteratively refine prompts and tools based on "personal best" and "global best" feedback found during the swarm search.

B.2 Benchmarks

Back to ToC

To evaluate the general-purpose task-solving capability of Mem^{2}Evolve, we conduct experiments across six task categories and eight benchmarks, as summarized in Table 6.

[8] We use the implementation at https://github.com/ryantzr1/OpenAlita as the official code is unavailable.

Table 6: **Overview of the benchmarks, domains, test set sizes, and evaluation metrics used in experiments.** † indicates that the test set was randomly sampled. ‡Average Score is the mean of Delivery Rate, Micro/Macro Commonsense Constraint Pass Rate, and Micro/Macro Hard Constraint Pass Rate.

Benchmark	Domain	Test Size	Metric
GAIA (Mialon et al., 2024)	General Assistant	166	Pass@1
ALFWorld (Shridhar et al., 2020)	Embodied Task	134	Success Rate
HotpotQA (Yang et al., 2018)	Multi-hop QA	500†	Exact Match (EM)
2WikiMultihopQA (Ho et al., 2020)	Multi-hop QA	500†	Exact Match (EM)
AIME 24	Math Reasoning	30	Pass@1
AIME 25	Math Reasoning	30	Pass@1
TravelPlanner (Xie et al., 2024)	Complex Planning	1,000	Average Score‡
WebShop (Yao et al., 2022a)	Web Navigation	251	Success Rate

- **GAIA**: GAIA is a benchmark designed to assess the capabilities of general-purpose AI assistants. It consists of 466 real-world, scenario-based questions covering daily tasks, scientific reasoning, web browsing, and tool usage. While these tasks are conceptually simple for humans, they remain challenging for advanced AI systems. We report Pass@1 as the primary evaluation metric.
- **ALFWorld**: ALFWorld aligns text-based games with embodied environments to evaluate an agent's ability to reason and act in interactive settings. It requires the agent to understand high-level goals and execute a sequence of low-level actions to interact with objects. We evaluate our method on 134 tasks and use Success Rate as the evaluation metric.
- **HotpotQA**: HotpotQA is a question-answering benchmark that challenges agents to perform multi-hop reasoning across multiple documents to locate relevant facts. In our experiments, we randomly sample a subset of 500 instances as the test set. Performance is evaluated using the Exact Match (EM) metric.
- **2WikiMultihopQA**: Similar to HotpotQA, 2WikiMultihopQA evaluates multi-hop reasoning capabilities over Wikipedia articles and features complex queries that require synthesizing information from multiple sources. We randomly sample 500 instances for testing and use EM for evaluation.
- **AIME 24/25**: These benchmarks correspond to the problem sets from the 2024 and 2025 American Invitational Mathematics Examinations. Each dataset consists of 30 high-difficulty problems that test advanced mathematical reasoning and creative problem-solving abilities. We use Pass@1 to measure accuracy.
- **TravelPlanner**: TravelPlanner evaluates language agents in complex tool-use and long-horizon planning scenarios, such as generating travel itineraries under strict constraints. We use the full test set of 1,000 instances. The final score is computed as the average of five metrics: Delivery Rate, Micro Commonsense Constraint Pass Rate, Macro Commonsense Constraint Pass Rate, Micro Hard Constraint Pass Rate, and Macro Hard Constraint Pass Rate.
- **WebShop**: WebShop is a simulated e-commerce environment containing over one million real-world products. It tests an agent's ability to navigate web pages, search, browse, and select options to fulfill user instructions. We evaluate on 251 test instances and measure performance using Success Rate.

B.3 RQ1: Ablation Study

Back to ToC

C Prompt Template

Prompt for Task Planning

Back to ToC

```
You are a task planning expert. Analyze the query carefully and break down complex queries into
logical, actionable steps.

USER QUERY: {query}

You MUST output EXACTLY ONE JSON object with this structure:
{
  "sub_tasks": [
    {
      "description": "Specific actionable task...",
      "dependencies": []
    },
    {
      "description": "Specific actionable task...",
      "dependencies": []
    }
  ]
}
RULES:
1. Each task description should be COMPLETE and SPECIFIC
2. Use task numbers (1,2,3...) for dependencies
3. First task usually has no dependencies: "dependencies": []
4. If task 2 depends on task 1: "dependencies": [1]
5. If task 3 depends on tasks 1 and 2: "dependencies": [1, 2]
6. Preserve the EXACT units and format requirements from the original query in task
descriptions
7. Output pure JSON format with no other content.
```

Prompt for Assess Tool Need

Back to ToC

```
You are a Principal Engineer. Your goal is to analyze the given multi-step task plan and
determine the most efficient way to solve it. Your primary objective is to use LLM-Native
capabilities and reuse existing tools. You will only determine that new tools are needed
(need_creation: true) as a last resort.


- Core Principles for Analysis
  - Maximize LLM-Native Capabilities (Highest Priority): You (the LLM) are the primary tool.
You can perform many tasks natively without code. Do NOT propose a tool for any step that
involves:
    - Reasoning, planning, or making decisions based on provided context.
    - Extracting, reformatting, filtering, sorting, or summarizing data from context.
    - Simple data transformations or list operations (e.g., finding an item, counting).
    - Any task that doesn't strictly require one of the "Why Code?" principles below.
  - Reuse First, Create Last (Second Priority): After exhausting LLM-Native capabilities, your
next goal is to reuse available_tools. Do NOT propose duplicates or near-duplicates.
  - Analyze the Full Plan: Review the entire sequence of steps to understand overall goals and
data flow.
  - Deconstruct Each Step: Mentally decompose each step into atomic required_capabilities.
  - Justify the Need for a Tool (Why Code?): A step requires a code tool only if it involves at
 least one of:
    - Complex Calculations: Non-trivial math, statistics, physics (beyond simple arithmetic).
    - Stateful Simulation/Iteration: Executing a loop many times while tracking changing state.
    - Complex Data Manipulation: Creating/modifying nested data structures according to precise
 rules (not simple extraction).
    - External I/O: Accessing APIs, files, databases, or system resources.
    - Deterministic Logic: When a precise, non-negotiable output is required based on complex
rules.

- Decision Procedure
```

```
  - Decompose Plan: Break down the plan into a flat list of atomic required_capabilities.
  - Evaluate Each Capability (3-step check):
      a. LLM-Native? Can this capability be handled directly by the LLM (based on Maximize LLM-
Native Capabilities)? If yes - it's covered.
      b. Existing Tool? If not LLM-native - can it be handled by one or more available_tools?
If yes - add those tool(s) to matching_tools.
      c. Gap Identified? Only if both above answers are NO - this capability is truly missing/
gap.

- Determine Final Output
      If all capabilities are covered by either above method:
        need_creation=false; missing_tools=[]
      Else if any essential gaps remain from above analysis:
        need_creation=true; propose minimal set of missing_tools covering ONLY those gaps

STRICT TYPE CONTRACT
- For every field listed below, emit values with the exact type specified.
- If you don't know a value, use an empty array [] or null where appropriate, never change the
type.
- Constraints per field (types in parentheses):
  - required_capabilities (string[]): flat list of atomic capabilities.
  - matching_tools (string[]): names of existing tools.
  - missing_tools (object[]): each item MUST be an object with fields:
    - name (string)
    - description (string)
    - reason (string)
    - example_input_output (string) - a single-line string formatted as: "Input: {...} ->
Output: {...}".
  - justification (string)
  - need_creation (boolean) - Phase 1 only: whether new tools are required.

Output Format

{
  "need_creation": true | false,
  "required_capabilities": ["verb_object", "..."],
  "matching_tools": ["existing_tool_a", "existing_tool_b"],
  "missing_tools": [
    {
      "name": "tool_for_gap",
      "description": "Does exactly this: ...",
      "reason": "Essential because: [Why Code? principle justification]",
      "example_input_output": "Input: {\"x\": 1} -> Output: {\"y\": 2}"
    }
  ],
  "justification": "1-3 sentences explaining overall coverage and gaps."
}

Validation Checklist
- matching_tools in current available tool names (exact match).
- missing_tools names are unique and non-empty; reasons reference the Why Code? principles.
- If existing tools fully cover all required_capabilities: need_creation=false and
missing_tools=[].
- Output must be a single JSON object with no extra commentary.
- Each tool does exactly one specific operation.

EXAMPLES OF GOOD TOOLS:
- `count_even_numbers` - counts even numbers in a list
- `extract_text_from_image` - extracts text from image (uses built-in vision)
- `calculate_percentage` - calculates percentage from two numbers
- `parse_addresses` - extracts and processes address data

EXAMPLES OF BAD TOOLS:
- `address_parser` + `parity_checker` + `sunset_awning_counter` (redundant!)
- `complex_data_analyzer` (too vague)
- `multi_purpose_processor` (does too many things)

Task:
```

```
{task}

Current Available Tools:
{available_tools}
```

Prompt for Tool Spec Creation

Back to ToC

```
You are a Principal Engineer responsible for writing implementable specifications for new tools.
Use the brief descriptions to generate detailed creation specs for ONLY those tools.

Design Principles (reused and specialized)
- One spec per missing tool; keep names consistent with Phase 1.
- Avoid overlapping with Current Available Tools; if overlap is detected, omit that spec.

STRICT TYPE CONTRACT
- For every field listed below, emit values with the exact type specified.
- If you don't know a value, use an empty array [] or null where appropriate, never change the
type.

Output Format:
{
  "tool_creation_specs": [
    {
      "tool_name": "tool_for_gap",
      "tool_description": "1-2 sentence summary.",
      "input_parameters": [
          { "name": "param", "type": "string", "description": "...", "required": true }
      ],
      "output_format": { "type": "object", "properties": { "result": { "type": "number" } } },
      "core_logic": [
        "Step 1: Input validation ...",
        "Step 2: Core processing ..."
      ]
    }
  ]
}

Task:
{task}

Current Available Tools:
{available_tools}
```

Prompt for Tool Creation

Back to ToC

```
You are a senior software engineering specialist building robust, reusable MCP (Model Context
Protocol) tools following Claude Desktop standards.

OBJECTIVE
Create a powerful and high-quality MCP tool to accurately and effectively solve the tasks
specified below. This tool should be a direct and effective solution to a given problem, and it
 should be generated strictly according to the **Core Implementation Logic**.

TASK
{original_query}

TOOL NAME
{tool_name}

Tool Description:
{tool_description}
```

```
Core Implementation Logic:
{core_logic}

Input Parameters:
{input_parameters}

Output Format:
{output_format}


DESIGN PRINCIPIPLES
- Solve the Problem Precisely: The tool's core goal is to provide a direct and effective
solution for the current task. Prioritize functional correctness and robustness over premature
generalization.
- Clear Interface Design: Clearly define the tool's input parameters and output format. Even if
 the tool has a specific function, its interface should be clear, well-documented, and easy to
use.
- Professional Naming Convention: Choose a descriptive, domain-prefixed snake_case name that
accurately reflects the tool's specific purpose.

OUTPUT CONTRACT (STRICT)
Return EXACTLY ONE JSON object as plain text (no markdown fences/backticks or extra prose). The
 object must have these keys:

1. name: string - professional general-purpose snake_case with domain prefix
   - Must be EXACTLY "{tool_name}" (do not rename or vary)

2. description: string - DETAILED description (5-10 sentences) covering:
   - What the tool does (core functionality)
   - Key capabilities and features
   - Typical use cases and scenarios
   - Input/output data types and formats
   - Any limitations or constraints

3. input_schema: object - JSON Schema (Claude Desktop standard) defining parameters:
   {{
     "type": "object",
     "properties": {{
       "param_name": {{
         "type": "string|number|boolean|array|object",
         "description": "Detailed parameter description",
         "enum": ["optional", "allowed", "values"],
         "default": "optional_default_value",
         "example": "input example"
       }}
     }},
     "required": ["list_of_required_params"]
   }}
   - Include ALL parameters the tool accepts
   - Provide detailed descriptions for each parameter
   - Specify types, constraints, enums, and defaults
   - Mark required vs optional parameters clearly

4. returns: object - Output format specification:
   {{
     "type": "string|object",
     "description": "Detailed description of return value",
     "format": "json|text|structured",
     "schema": {{"optional": "output schema for structured returns"}}
   }}

5. module_code: string - COMPLETE Python module source to be saved as `<tool_name>.py`

Module requirements (STRICT):
- Include necessary imports.
- Define ONE public function implementing the tool with an EXPLICIT parameter list derived from
 input_schema (no **kwargs).
```

```
- Implement robust validation and error handling per input_schema (types, required fields,
enums, ranges).
- Provide clear control flow and helpful error messages referencing parameter names.
- Define TOOL_CONFIG = {{ "name": "{tool_name}", "description": <desc>, "function": <
function_object>, "input_schema": <schema>, "returns": <returns> }} at module scope.
- Multi-line Python code with normal 4-space indentation.
- No external network calls or dependencies unless they are clearly documented and optional.

Naming constraints (STRICT):
- The JSON field name MUST equal "{tool_name}".
- In the Python module, TOOL_CONFIG['name'] MUST equal "{tool_name}".
- Assume the file will be saved as "{tool_name}".py; do not reference any other module name.

Formatting constraints:
- No markdown fences.
- No trailing backslashes at line ends.
- The module must be self-contained and importable.

CAPABILITIES & QUALITY
- Robust Input Handling: The tool must robustly handle various expected inputs and provide
clear, helpful error messages for invalid or edge-case inputs.
- Graceful Degradation: Provide graceful degradation when inputs or external resources (if any)
 are missing or invalid.
- Strict Validation: Validate all inputs against the input_schema at the beginning of the
function.
- Code Excellence: Optimize for clarity, maintainability, and performance; follow PEP 8
standards.
- Helpful Errors: Add helpful error messages that reference parameter names to guide the user.

REMINDERS
- Output must be a single JSON object (no additional commentary).
- Absolutely no json or python fences in the output.
- Ensure input_schema follows JSON Schema specification exactly.
- Provide detailed, production-ready documentation in all fields.
```

Prompt for Agent Creation

Back to ToC

```
You are an expert agent designer. Create a minimal, reusable specialist agent specification for
 the task.

Inputs
- User Task: {sub_task}
- Available Tools: {tools}

Output format
- Return exactly one JSON object as the entire response.
- The object must contain only these four keys, in this exact order: role, suggestions, tools,
expertise.
- Explicit schema:
  - role: string
  - suggestions: array of 3-5 strings
  - tools: array of 0-3 strings (tool names from Available Tools)
  - expertise: string (1-2 sentences; concise domain strengths and typical methodology)
- Do not include any other text, comments, code fences, or fields.

Constraints
Role
- A clear, human-readable professional title suitable for a business card.
- 2-5 words, Title Case, English only, no emojis/symbols.

Suggestions
- An array of 3-5 short, concrete execution hints for approaching this class of tasks.
- Each item starts with an imperative verb and is 6-16 words.
- Specific within the domain implied by {sub_task}, yet general-purpose (not tailored to a
single query).
```

```
- No placeholders, no meta references (e.g., "JSON", "this prompt"), avoid tool names unless
essential to the method.

Tools
- Choose a minimal subset of tool names taken verbatim from Available Tools.
- Use exact casing/spelling; do not invent, modify, or describe tools; no duplicates.
- If no tool is applicable or Available Tools is empty, use an empty array [].

Expertise
- 1-2 sentences (20-160 characters) summarizing domain strengths and typical methodology.
- Mention information gathering, evaluation, analysis (quantitative/qualitative as applicable),
 and synthesis.
- Avoid listing tools; focus on capabilities and working approach.

General
- Ensure the role, suggestions, tools, and expertise are coherent with one another and with
{sub_task}.
- Do not echo the inputs. Do not add explanations.
- Output must be valid JSON using double quotes and no trailing commas.
```

Prompt for React Template

Back to ToC

```
You are a {role}. Based on prior agents' results and completed steps, your goal is to complete
the current task as effectively as possible.

# Suggestions
{suggestions}

# Progress So Far
{previous}

# Past Experience References
Below are relevant experiences from previous similar tasks. Reference the success experiences
to learn effective approaches, and learn from the failure experiences to avoid common pitfalls.

## Success Experiences
{success_experiences}

## Failure Experiences
{failure_experiences}

You have access to the following tools:
# Tools
{tool}

# Steps
1. Review the outputs from previous agents to understand progress and context.
2. Analyze the task and decompose it if needed. Utilize the available tools as appropriate.
3. Define the current step you will complete, labeling it as 'CurrentStep'.
4. Choose one Action from the available tools to execute the current step.

# Format example
{format_example}
```

Prompt for LLM as a Judge

Back to ToC

```
You are a Judge LLM responsible for evaluating the entire execution trajectory of a task and
determining whether it was completed correctly.

OBJECTIVE
Analyze the complete task execution trajectory and provide a comprehensive evaluation of task
```

```
completion, agent performance, and tool effectiveness.

TRAJECTORY INFORMATION

Original Query:
{query}

Decomposed Subtasks:
{subtasks}

Agent Assignments:
{agent_assignments}

Agent Execution Processes:
{agent_processes}

Result Aggregation Process:
{aggregation_process}

Final Result:
{final_result}

Newly Created Tools (if any):
{created_tools}

EVALUATION CRITERIA

1. Task Completion Assessment:
   - Does the final result correctly answer the original query?
   - Are all decomposed subtasks properly addressed?
   - Is the aggregated result logically coherent and complete?

2. Agent Performance Assessment:
   - Did each agent execute its assigned subtask correctly?
   - Were the agents' reasoning processes sound and effective?
   - Did agents properly utilize available tools?

3. Tool Effectiveness Assessment:
   - Did existing tools function correctly when used?
   - Were newly created tools (if any) implemented correctly?
   - Did tools produce expected outputs?

EVALUATION OUTPUT FORMAT

You must provide your evaluation in the following JSON format:

{{
  "task_completed": true/false,
  "completion_quality": "good/poor",
  "overall_assessment": "Detailed overall assessment of task completion",
  "agent_evaluations": [
    {{
      "agent_id": "agent identifier",
      "agent_role": "role name",
      "subtask_id": "subtask identifier",
      "performance": "success/failure",
      "strengths": ["strength 1", "strength 2", ...],
      "issues": ["issue 1", "issue 2", ...]
      }},
    ...
  ],
  "tool_evaluations": [
    {{
      "tool_name": "tool name",
      "tool_type": "existing/newly_created",
      "effectiveness": "success/partial_success/failure",
      "issues": ["issue 1", "issue 2", ...],
    }},
```

```
    ...
  ]
}}

IMPORTANT GUIDELINES

1. Be objective and thorough in your evaluation
2. Provide specific, actionable feedback
3. Identify both strengths and weaknesses
4. Focus on patterns that can inform future improvements
5. Distinguish between agent failures and tool failures
6. Consider the complexity of the task when evaluating performance
7. Ensure all JSON is properly formatted and valid

Return ONLY the JSON evaluation, no additional text.
```

Prompt for Tool Memory Generation

Back to ToC

```
You are a technical documentation specialist creating comprehensive tool implementation guides.

OBJECTIVE
Generate a detailed markdown section that captures the implementation experience of this tool
for future reference and reuse. This will be one entry within a broader topic file.

TOOL INFORMATION
Tool Name: {tool_name}
Tool Description: {tool_description}
Required Capabilities: {required_capabilities}
Usage Examples: {usage_examples}

Tool Implementation Code:
{tool_code}

DOCUMENT STRUCTURE
Create a markdown section with the following structure:

## [Generate a specific "How to..." question title here]

The title should:
- Start with "How to" or "How can I"
- Be specific to what this exact implementation does
- Be 5-15 words long
- Clearly distinguish this implementation from similar ones
- Example: "How to Download Files From a URL?", "How to Parse CSV Files with Custom Delimiters
?"

### Description
A concise 2-3 sentence overview explaining what this specific implementation does and what
problem it solves.

### Use Cases
A bulleted list of practical scenarios where this implementation is applicable. Include:
- Specific real-world situations
- Types of data or inputs it handles
- Common integration patterns

### Code Implementation
The core implementation of the tool. Format as:
‘‘‘python
[Include the complete, clean tool implementation code here]
‘‘‘

### Tool Configuration
(Optional - only include if the tool uses API keys, URLs, or other configuration)
Document any configuration requirements:
```

```
- API keys needed
- Environment variables
- URL endpoints
- Configuration parameters

QUALITY REQUIREMENTS
1. Write in clear, professional technical documentation style
2. Use proper markdown formatting (## for main title, ### for subsections)
3. Be concise but comprehensive
4. Focus on reusability and understanding
5. Include practical context, not just code
6. Highlight key implementation patterns
7. Note any important dependencies or requirements

OUTPUT FORMAT
Return ONLY the complete markdown section. Do not include any preamble, explanations, or
commentary outside the document itself. The section MUST start with "## How to..." as the first
 line.
```

Prompt for Success Agent Memory Generation

Back to ToC

```
You are an experience synthesis specialist creating memory entries for an agent based on
successful task execution.

OBJECTIVE
Generate a comprehensive memory entry that captures the successful experience of this agent,
including successful strategies, effective tool usage, and valuable insights for future similar
 tasks.

Agent Role: {agent_role}
Agent Skills: {agent_skills}

TASK EXECUTION INFORMATION
Subtask Description: {subtask_description}
Task Context: {task_context}
Tools Used: {tools_used}
Execution Process:
{execution_process}

Final Result: {final_result}

JUDGE EVALUATION
Performance Rating: {performance_rating}
Strengths Identified: {strengths}
Key Success Patterns: {success_patterns}

MEMORY ENTRY STRUCTURE

Generate a memory entry in the following markdown format:

## [Create a specific, descriptive title for this experience]

The title should:
- Be specific to the type of task handled
- Highlight the key capability demonstrated
- Be 5-15 words long
- Example: "How to Extract and Summarize Information from Multiple Web Sources"

### Description
A concise 2-3 sentence overview of what this experience teaches about handling this type of
task.

### Task Context
Describe the original problem and the conditions under which this task was performed.
```

```
### Experience of Success
Detail the successful strategy and reasoning process:
- **Successful Strategy**: Key decision points, problem-solving approach, and how tools were
selected
- **Tools and Techniques**: Tools used and how they contributed to success
- **Key Insights**: What made this approach effective and when to apply similar strategies
- **Applicable Scenarios**: Similar task types where this experience would be valuable
- **Potential Pitfalls**: Conditions that might cause similar approaches to fail and warning
signs to watch for

QUALITY REQUIREMENTS
1. Write in clear, professional documentation style
2. Use proper markdown formatting
3. Be specific and actionable
4. Focus on transferable knowledge
5. Highlight decision-making processes
6. Include both what worked and why it worked

OUTPUT FORMAT
Return ONLY the complete markdown memory entry. Do not include any preamble or explanations
outside the memory entry itself. The entry MUST start with "##" as the first line.
```

Prompt for Failure Agent Memory Generation

Back to ToC

```
You are an error analysis specialist creating memory entries that help agents learn from
failures.

OBJECTIVE
Generate a comprehensive memory entry that captures the failure experience, analyzes root
causes, and provides clear guidance on how to avoid similar failures in the future.

AGENT INFORMATION
Agent Role: {agent_role}
Agent Skills: {agent_skills}

TASK EXECUTION INFORMATION
Subtask Description: {subtask_description}
Task Context: {task_context}
Tools Attempted: {tools_used}
Execution Process:
{execution_process}

Failure Outcome: {failure_outcome}

JUDGE EVALUATION
Performance Rating: {performance_rating}
Issues Identified: {issues}
Root Cause Analysis: {root_cause}
Suggested Fixes: {suggested_fixes}

MEMORY ENTRY STRUCTURE

Generate a memory entry in the following markdown format:

## [Create a specific, descriptive title for this failure case]

The title should:
- Clearly indicate the problematic scenario
- Be specific enough to match similar future situations
- Be 5-15 words long
- Example: "Common Pitfalls When Parsing Inconsistent CSV Data Formats"

### Description
A concise 2-3 sentence overview of what went wrong and the key lesson to take away.
```

```
### Task Context
Describe the task and conditions that led to this execution:
- What was attempted
- What was expected
- What assumptions were made

### Experience of Failure
Detail what went wrong and how to avoid it:
- **What Went Wrong**: Specific errors, incorrect decisions, tools misused, or logic errors
- **Root Cause Analysis**: Why the approach didn't work, what was misunderstood, gaps in
knowledge
- **Corrective Actions**: Alternative approaches, additional checks, tools to use instead,
validation steps
- **Warning Signs**: Task characteristics that trigger this issue, context patterns, red flags
during execution
- **Related Success Patterns**: Successful approaches that should be used instead
- **Partial Successes**: Parts of the approach that were correct, tools or techniques that
functioned properly

QUALITY REQUIREMENTS
1. Write in clear, instructive style
2. Use proper markdown formatting
3. Be honest and specific about failures
4. Provide actionable corrective guidance
5. Focus on learning and improvement
6. Help prevent similar failures

OUTPUT FORMAT
Return ONLY the complete markdown memory entry. Do not include any preamble or explanations
outside the memory entry itself. The entry MUST start with "##" as the first line.
```

D Case Study

D.1 Tool Implementation for *Simulate Piston Platform Game*

Back to ToC

As shown in Code 1, when performing complex probabilistic reasoning tasks, the agent extends its modeling capabilities by constructing a simulation tool and employing Monte Carlo methods to estimate the true underlying probabilities. The implementation of this tool is entirely grounded in the original problem formulation and faithfully simulates all rules specified in the task. Moreover, the implementation exhibits strong robustness, supporting simulations with arbitrary numbers of trials and arbitrary numbers of balls, and is, to some extent, adaptable to multiple variants of the original problem.

```python
import random

def simulate_piston_platform_game(num_balls=100, num_simulations=100000):
    """
    Simulates a specific ping-pong game mechanism involving a ball queue, a limited-capacity platform,
    and three random pistons with complex ejection/replacement rules. Used to calculate the
    probability of each ball number being 'ejected by a piston' (winning) versus being 'released' (eliminated).

    Args:
        num_balls (int): Total number of balls in the queue, numbered 1 to N.
        num_simulations (int): Monte Carlo iterations.

    Returns:
        dict: A dictionary containing 'win_probabilities' and 'best_choice'.
    """
    # Step 1: Validate inputs
    if not isinstance(num_balls, int) or num_balls < 1:
        raise ValueError(f"Input 'num_balls' must be a positive integer. Got: {num_balls}")

    if not isinstance(num_simulations, int) or num_simulations < 1:
        raise ValueError(f"Input 'num_simulations' must be an integer >= 1. Got: {num_simulations}")

    # Step 2: Initialize win_counts
    # Keys are integers 1 to N, initialized to 0
    win_counts = {i: 0 for i in range(1, num_balls + 1)}

    # Step 3: Begin simulation loop
    for _ in range(num_simulations):
        # Step 3a: Initialize queue and platform
        deck = list(range(1, num_balls + 1))
        platform = []

        # Initial fill of the platform (max 3 balls)
        for _ in range(3):
            if deck:
                platform.append(deck.pop(0))
            else:
                platform.append(None)

        # Helper function to refill balls from deck
        def get_next_balls(count):
            new_balls = []
            for _ in range(count):
                if deck:
                    new_balls.append(deck.pop(0))
                else:
                    new_balls.append(None)
            return new_balls

        # Step 4: Simulate piston firing process
        # Continue while there are still balls on the platform
        while any(ball is not None for ball in platform):
            # Select a random piston (1, 2, or 3)
```

```
            piston = random.randint(1, 3)

            # Current platform state
            p1, p2, p3 = platform[0], platform[1], platform[2]

            # Step 5: Apply Piston Rules
            if piston == 1:
                # If Piston 1: Eject Pos 1 (Win). Move Pos 2->1, Pos 3->2. Refill 1 ball.
                if p1 is not None:
                    win_counts[p1] += 1

                incoming = get_next_balls(1)
                platform = [p2, p3, incoming[0]]

            elif piston == 2:
                # If Piston 2: Eject Pos 2 (Win). Release Pos 1 (Loss). Move Pos 3->1. Refill 2 balls.
                if p2 is not None:
                    win_counts[p2] += 1

                # p1 is released (eliminated), so we don't increment its win count

                incoming = get_next_balls(2)
                platform = [p3, incoming[0], incoming[1]]

            elif piston == 3:
                # If Piston 3: Eject Pos 3 (Win). Release Pos 1 (Loss). Move Pos 2->1. Refill 2 balls.
                if p3 is not None:
                    win_counts[p3] += 1

                # p1 is released (eliminated), so we don't increment its win count

                incoming = get_next_balls(2)
                platform = [p2, incoming[0], incoming[1]]

    # Step 8: Calculate probabilities
    win_probabilities = {}
    for i in range(1, num_balls + 1):
        win_probabilities[str(i)] = win_counts[i] / num_simulations

    # Find the ball with the highest probability
    best_ball_num = max(win_counts, key=win_counts.get)

    # Step 9: Format output object
    return {
        "win_probabilities": win_probabilities,
        "best_choice": best_ball_num
    }

```

Code 1: Code Implenmenation for Simulate Piston Platform Game Tool

D.2 Tool Implementation for *Youtube Audio Transcriber*

Back to ToC

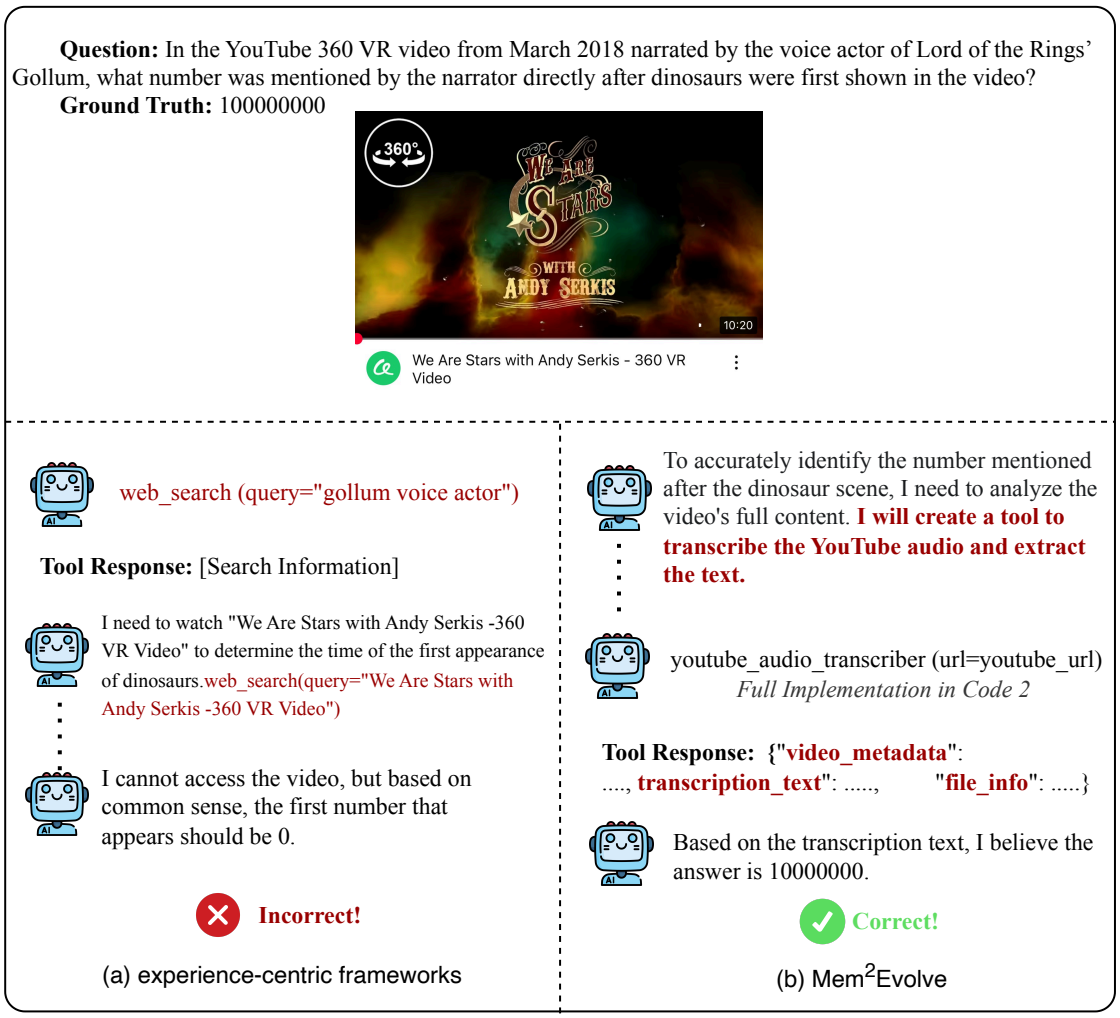


Figure 7: **Case Study 2 on YouTube Video Subtitle Extraction.** When initialized with only a web search tool, (a) experience-centric frameworks fail to handle tasks situated beyond their capability boundary, such as retrieving internal video content, leading to incorrect answers based on general common sense. In contrast, (b) Mem^{2}Evolve leverages the guidance of accumulated experience to dynamically generate high-quality tools (e.g., a custom subtitle transcriber), effectively breaking through capability boundaries to access the necessary context and derive the correct answer.

As illustrated in Code 2, when the model encounters complex tasks in open-world scenarios, such as "In the YouTube 360 VR video narrated by the voice actor of Gollum from *The Lord of the Rings* in March 2018, what number does the narrator directly mention after the dinosaur first appears?", it is necessary to perform semantic analysis on the audio content of YouTube videos to obtain the correct answer. To address such problems, the Agent autonomously generates the *Youtube Audio Transcriber* tool, which integrates open-source libraries including yt-dlp, pytube, and speech_recognition to enable audio transcription for arbitrary YouTube video URLs with multi-language support, thereby generalizing to similar tasks. The design of this tool demonstrates three key characteristics: First, the implementation leverages mature open-source components rather than building from scratch, significantly improving development efficiency and reliability. Second, the tool exhibits strong robustness, effectively handling diverse input formats and various edge cases. Third, the tool adheres to the principle of generality, targeting a category of tasks rather than a single problem, thus possessing good reusability to support the execution of subsequent similar tasks.

```python
import os
import math
import shutil
import json
from urllib.parse import urlparse, parse_qs
import yt_dlp
from pydub import AudioSegment
import speech_recognition as sr


def youtube_audio_transcriber(youtube_url: str, language_code: str = "en-US", chunk_duration_sec: int = 15):
    """
    Downloads audio from YouTube and transcribes it into text.

    Args:
        youtube_url (str): The full URL of the YouTube video.
        language_code (str): Language code for recognition (default 'en-US').
        chunk_duration_sec (int): Duration of each processing chunk in seconds (default 15).

    Returns:
        dict: Object containing transcription_text, video_metadata, and file_info.
    """

    workspace_dir = "workspace"
    temp_dir = "temp_chunks"

    # Ensure directories exist
    if not os.path.exists(workspace_dir):
        os.makedirs(workspace_dir)

    video_id = None
    downloaded_mp3_path = None
    full_transcription = ""
    video_metadata = {}

    try:
        # --- Step 1: Validate inputs and parse the `youtube_url` to extract the Video ID ---
        if 'youtube.com' in youtube_url:
            parsed_url = urlparse(youtube_url)
            video_id = parse_qs(parsed_url.query).get('v', [None])[0]
        elif 'youtu.be' in youtube_url:
            parsed_url = urlparse(youtube_url)
            video_id = parsed_url.path.lstrip('/')

        if not video_id:
            raise ValueError(f"Could not extract Video ID from URL: {youtube_url}")

        # --- Step 2: Configure `yt_dlp` options ---
        # Set output template to workspace, convert to mp3 192kbps
        output_template = os.path.join(workspace_dir, f"{video_id}_%(title)s.%(ext)s")

        ydl_opts = {
            'format': 'bestaudio/best',
            'outtmpl': output_template,
            'postprocessors': [{
                'key': 'FFmpegExtractAudio',
                'preferredcodec': 'mp3',
                'preferredquality': '192',
            }],
            'quiet': True,
            'no_warnings': True,
        }

        # --- Step 3: Execute the download & Extract Metadata ---
        print(f"[Tool] Starting download for Video ID: {video_id}...")
```

```
        with yt_dlp.YoutubeDL(ydl_opts) as ydl:
            info = ydl.extract_info(youtube_url, download=True)

            # Format duration string
            duration = info.get('duration', 0)
            m, s = divmod(duration, 60)
            h, m = divmod(m, 60)
            duration_str = f"{h:02d}:{m:02d}:{s:02d}" if h else f"{m:02d}:{s:02d}"

            video_metadata = {
                "title": info.get('title', 'Unknown'),
                "uploader": info.get('uploader', 'Unknown'),
                "duration_str": duration_str,
                "video_id": video_id
            }

        # --- Step 4: Locate the downloaded MP3 file ---
        # yt-dlp might replace characters in the filename, so we search by video_id
        for file in os.listdir(workspace_dir):
            if video_id in file and file.endswith('.mp3'):
                downloaded_mp3_path = os.path.join(workspace_dir, file)
                break

        if not downloaded_mp3_path:
            raise FileNotFoundError("Audio file not found after download process.")

        # --- Step 5: Initialize SpeechRecognition & Temp Directory ---
        recognizer = sr.Recognizer()

        if os.path.exists(temp_dir):
            shutil.rmtree(temp_dir)
        os.makedirs(temp_dir)

        # --- Step 6: Load MP3 & Calculate Chunks ---
        print("[Tool] Loading audio file for processing...")
        audio = AudioSegment.from_file(downloaded_mp3_path)

        # pydub works in milliseconds
        chunk_length_ms = chunk_duration_sec * 1000
        num_chunks = math.ceil(len(audio) / chunk_length_ms)

        print(f"[Tool] Audio length: {len(audio)/1000:.2f}s. Split into {num_chunks}
    chunks.")

        # --- Step 7, 8, 9: Iterate, Slice, Recognize, Append ---
        print("[Tool] Starting transcription...")

        for i in range(num_chunks):
            start_ms = i * chunk_length_ms
            end_ms = (i + 1) * chunk_length_ms

            # Slice audio
            chunk = audio[start_ms:end_ms]

            # Export to WAV (required by SpeechRecognition)
            chunk_filename = os.path.join(temp_dir, f"chunk_{i}.wav")
            chunk.export(chunk_filename, format="wav")

            # Recognize
            with sr.AudioFile(chunk_filename) as source:
                audio_data = recognizer.record(source)
                try:
                    text = recognizer.recognize_google(audio_data, language=
language_code)
                    full_transcription += text + " "
                except sr.UnknownValueError:
                    # Audio was not understood (silence, noise, music)
                    pass
                except sr.RequestError as e:
                    print(f"[Tool] API Error on chunk {i}: {e}")
```

```
                except Exception as e:
                    print(f"[Tool] Unexpected error on chunk {i}: {e}")

        full_transcription = full_transcription.strip()

        # Get file size for report
        file_size_mb = os.path.getsize(downloaded_mp3_path) / (1024 * 1024)

        # --- Step 10: Clean up temporary files ---
        if os.path.exists(temp_dir):
            shutil.rmtree(temp_dir)

        # --- Step 11: Return result object ---
        return {
            "transcription_text": full_transcription,
            "video_metadata": video_metadata,
            "file_info": {
                "local_path": downloaded_mp3_path,
                "file_size_mb": round(file_size_mb, 2)
            }
        }

    except Exception as e:
        # Cleanup temp if error occurs
        if os.path.exists(temp_dir):
            shutil.rmtree(temp_dir)

        # Return error structure or raise
        return {
            "error": str(e),
            "transcription_text": "",
            "video_metadata": video_metadata if video_metadata else {},
            "file_info": {}
        }

```

Code 2: Tool Implenmenation for Youtube Audio Transcriber

D.3 Experience Guidance Tool Creation

Back to ToC

In this case study, we demonstrate how Mem^{2}Evolve creates a new tool under the guidance of Experience Memory. When the system already contains a memory item titled "***How to Analyze Images Using the GPT-4o Multimodal Model?***" (Figure 8), the model, in generating a tool (Code 3) for analyzing YouTube video content, first extracts screenshots from the video at fixed frame intervals. Subsequently, it applies GPT-4o to analyze each extracted frame and automatically constructs a complete, well-aligned prompt tailored to the analysis task.

```
import os
import json
import subprocess
import base64
import requests
import math
import shutil
import tempfile
from dotenv import load_dotenv

load_dotenv()


def analyze_video_for_species(youtube_url: str, target_subject: str = "bird species"
    , sampling_interval: int = 10):
    """
    Analyzes a YouTube video to determine the maximum number of distinct species
    visible simultaneously.
    """

    api_key = os.getenv("OPENAI_API_KEY")
```

How to Analyze Images Using GPT-4o Multimodal Model?

Description
Parse and analyze an image file using GPT-4o multimodal model. This code can understand complex visual content, generate captions, extract tables as HTML, create SVG code for geometric shapes, and answer specific questions about images.

Use Cases
- Product image analysis for e-commerce catalog management
- Medical image interpretation and diagnostic support
- Security and surveillance image analysis
- Educational content creation from visual materials
- Art and cultural artifact documentation
- Scientific image analysis and research documentation
- Social media content moderation and analysis

Tool Implementation
```python
# Partial code implementation is omitted here

# Prepare API request payload
payload = {
    "model": "gpt-4o-2024-11-20",
    "messages": [
        {
            "role": "user",
            "content": [
                {
                    "type": "text",
                    "text": prompt,
                },
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"{img_type}{img_base64}"
                    }
                }
            ],
        },
    ],
    "max_tokens": 16384,
}

# Get API credentials from environment variables
api_key = os.getenv("OPENAI_API_KEY")
api_base = os.getenv("OPENAI_BASE_URL")

headers = {
    "Content-Type": "application/json",
    "Authorization": f"Bearer {api_key}"
}

# Send request to OpenAI API
response = requests.post(f"{api_base}/chat/completions", headers=headers, json=payload)

result = response.json()
output = result["choices"][0]["message"]["content"]
```

Figure 8: **Tool Experience: Using the GPT-4o for Image Analysis.** This tool experience illustrates how to call the GPT-4o API to analyze images, where the agent can customize prompts to steer GPT-4o toward diverse and complex visual understanding tasks (e.g., recognition, counting, spatial reasoning, chart/diagram interpretation, and multimodal grounding). Each tool experience is organized into four fields: *Title*, *Description*, *Use Cases*, and *Tool Implementation*.

```
    if not api_key:
        return {"error": "Missing OPENAI_API_KEY environment variable."}

    workspace_dir = "workspace"
    if not os.path.exists(workspace_dir):
        os.makedirs(workspace_dir)

    max_species_count = 0
    best_frame_data = {
        "count": 0,
        "species_list": [],
        "description": "No data found."
    }
    best_timestamp = "00:00:00"

    try:
        # --- Step 1: Validate URL and Extract Metadata ---
        print(f"[Tool] Getting video info for: {youtube_url}")
        cmd_info = ['yt-dlp', '--dump-json', '--no-playlist', youtube_url]
        result_info = subprocess.run(cmd_info, capture_output=True, text=True,
    timeout=30)

        if result_info.returncode != 0:
            raise ValueError(f"Failed to extract video info: {result_info.stderr}")

        video_info = json.loads(result_info.stdout)
        duration = video_info.get('duration') # seconds
        video_id = video_info.get('id', 'unknown')

        if not duration:
            raise ValueError("Could not determine video duration.")

        # --- Step 2: Calculate Timestamps ---
        # Limit total checks to avoid excessive API usage cost in this demo
    implementation
        # For production, you might want to remove the limit or increase interval
        timestamps_sec = range(0, int(duration), sampling_interval)
        total_steps = len(timestamps_sec)

        print(f"[Tool] Video Duration: {duration}s. Sampling every {
    sampling_interval}s. Total checks: {total_steps}")

        # --- Step 3 & 4: Iterate through timestamps ---
        for idx, current_sec in enumerate(timestamps_sec):

            # Format timestamp HH:MM:SS
            m, s = divmod(current_sec, 60)
            h, m = divmod(m, 60)
            timestamp_str = f"{h:02d}:{m:02d}:{s:02d}"

            print(f"[Tool] ({idx+1}/{total_steps}) Processing timestamp: {
    timestamp_str}...")

            # --- Step 5: Download Segment (using ffmpeg downloader for speed) ---
            # Create a unique temp file for this segment
            temp_segment_path = os.path.join(workspace_dir, f"temp_{video_id}_{
    current_sec}.mp4")
            screenshot_path = os.path.join(workspace_dir, f"frame_{video_id}_{
    current_sec}.jpg")

            try:
                # Use yt-dlp with ffmpeg external downloader to fetch just a tiny
    snippet
                # This avoids downloading the whole video
                download_cmd = [
                    'yt-dlp',
                    '--format', 'best[height<=720]', # 720p is enough for
    recognition
                    '--external-downloader', 'ffmpeg',
                    '--external-downloader-args', f'ffmpeg_i:-ss {current_sec} -t 2'
```

```
    , # download 2 seconds
                '--output', temp_segment_path,
                '--quiet', '--no-warnings',
                youtube_url
            ]
            subprocess.run(download_cmd, capture_output=True, timeout=60)

            # --- Step 6: Extract Screenshot ---
            # Check if video segment exists (sometimes yt-dlp appends ext)
            found_video = None
            for ext in ['.mp4', '.webm', '.mkv']:
                check_path = temp_segment_path.replace('.mp4', ext) # naive replacement
                if os.path.exists(check_path): # Check exact match first if template wasn't dynamic
                    found_video = check_path
                    break
                # Handle yt-dlp output template behavior if needed
                if os.path.exists(temp_segment_path):
                    found_video = temp_segment_path
                    break

            if not found_video:
                print(f"    -> Warning: Could not download segment for {timestamp_str}, skipping.")
                continue

            # Capture first frame of the segment
            ffmpeg_cmd = [
                'ffmpeg', '-i', found_video,
                '-vframes', '1', '-q:v', '2', '-y',
                screenshot_path
            ]
            subprocess.run(ffmpeg_cmd, capture_output=True, timeout=10)

            if not os.path.exists(screenshot_path):
                print(f"    -> Warning: Screenshot failed for {timestamp_str}, skipping.")
                continue

            # --- Step 7: Encode Base64 ---
            with open(screenshot_path, "rb") as image_file:
                base64_image = base64.b64encode(image_file.read()).decode('utf-8')

            # --- Step 8: Construct AI Prompt ---
            prompt_text = (
                f"Analyze this image specifically to count {target_subject}. "
                f"Identify all distinct {target_subject} present. "
                "Ignore statues, pictures, or reflections if clearly not real/live instances (unless the target is inanimate). "
                "Return ONLY valid JSON with the following keys: "
                "'count' (integer, number of UNIQUE species visibly distinct simultaneously), "
                "'species_list' (array of strings, names of identified species), "
                "'description' (string, brief summary of the scene)."
            )

            payload = {
                "model": "gpt-4o",
                "messages": [
                    {
                        "role": "user",
                        "content": [
                            {"type": "text", "text": prompt_text},
                            {
                                "type": "image_url",
                                "image_url": {"url": f"{base64_image}"}
```

```
                                }
                            ]
                        }
                    ],
                    "max_tokens": 500,
                    "response_format": {"type": "json_object"}
                }

                # --- Step 9: Call AI API ---
                headers = {
                    "Content-Type": "application/json",
                    "Authorization": f"Bearer {api_key}"
                }

                response = requests.post("https://api.openai.com/v1/chat/completions", headers=headers, json=payload)
                response_json = response.json()

                if "choices" not in response_json:
                    print(f"    -> API Error: {response_json}")
                    continue

                content_str = response_json['choices'][0]['message']['content']
                analysis = json.loads(content_str)

                current_count = analysis.get('count', 0)
                print(f"    -> Found: {current_count} species ({analysis.get('species_list')})")

                # --- Step 10: Compare and Update ---
                if current_count > max_species_count:
                    max_species_count = current_count
                    best_timestamp = timestamp_str
                    best_frame_data = analysis
                    print(f"    -> New Record! {max_species_count} at {best_timestamp}")

            except Exception as e:
                print(f"    -> Error processing step {timestamp_str}: {e}")

            finally:
                # --- Step 11: Cleanup ---
                if os.path.exists(screenshot_path):
                    os.remove(screenshot_path)
                if 'found_video' in locals() and found_video and os.path.exists(found_video):
                    os.remove(found_video)

        # --- Step 12: Return Results ---
        result = {
            "max_simultaneous_species": max_species_count,
            "best_timestamp": best_timestamp,
            "identified_species": best_frame_data.get("species_list", []),
            "analysis_summary": best_frame_data.get("description", "")
        }

        return result

    except Exception as e:
        return {"error": str(e)}

```

Code 3: Code Implenmenation for Analyze Video for Species Tool

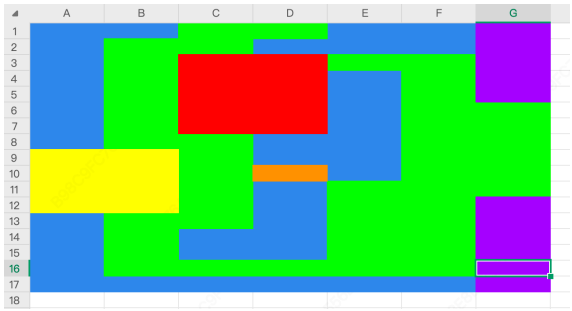
D.4 Comparison of Tool Generation With and Without Experience Guidance

Back to ToC

This case provides an intuitive demonstration of the guiding role of Experience Memory in the generation of new tools. Specifically, as shown in Figure 9, Mem^{2}Evolve is required to read color-coded cells from an Excel file during task execution. With guidance from experience, the tool generated by Mem^{2}Evolve

Question: Each cell in the attached spreadsheet represents a plot of land. **The color of the cell indicates who owns that plot. Green cells are plots owned by Earl Smith.** Can Earl walk through every plot he owns (and no other plots) and return to his starting plot without backtracking? For this question, consider backtracking to be any instance where Earl would enter a plot of land he had already entered since leaving his starting plot.

File Content:



Ground Truth: No

read_excel_basic (file_path=file_path)
Full Implementation in Code 4

Tool Response: {'file_name': '5cfb274c-0207-4aa7-9575-6ac0bd95d9b2.xlsx', 'sheet_content': {'Sheet1': ""}'}

I don't have enough information!

Incorrect!

(a) w/o experience-guidance

parse_excel_with_styles (file_path=file_path)
Full Implementation in Code 3

Tool Response: ..., <table border='1' style='border-collapse:collapse;'> <tr><th style='background-color:#4A86E8;'></th><th style='background-color:#4A86E8;'>, ...

Based on the information in the attachment, I believe the answer is No.

Correct!

(b) w/ experience-guidance

Figure 9: **Case Study 4: Experience-Guided Tool Generation for Attribute-Preserving Excel Parsing.** This case illustrates how Experience Memory guides Mem^{2}Evolve to generate task-appropriate tools that preserve critical non-textual attributes. When required to extract color-coded cells from an Excel file, Mem^{2}Evolve leverages past experience to synthesize a tool capable of accurately retrieving both cell values and their original color information in a standardized output format (full implementation in Code 4). In contrast, without experiential guidance, the generated tool relies solely on pandas, which fails to retain color attributes (full implementation in Code 5), leading to unsuccessful task execution.

is able to accurately retrieve the target cells together with their original color information and output the results in a standardized format (full implementation in Code 4). In contrast, in the absence of such experiential guidance, the model generates a tool that relies solely on pandas to read the Excel content, failing to preserve and return the color attributes (full implementation in Code 5). This limitation ultimately leads to unsuccessful task execution.

```python
import os
import pandas as pd
from openpyxl import load_workbook

def parse_excel_with_styles(file_path: str, row_limit: int = 100):
    """
    Parses an Excel or CSV file and returns the content as formatted HTML with style information preserved.
    """

    # Internal helper function to extract styles
    def get_cell_style(cell):
        """Extract style information from a cell and return as CSS style string."""
        styles = []

        # Check for bold formatting
        if cell.font and cell.font.bold:
            styles.append('font-weight:bold;')

        # Check for italic formatting
        if cell.font and cell.font.italic:
            styles.append('font-style:italic;')

        # Extract font color
        # Step 6 & 7: Handle Color Processing (ARGB -> RGB)
        color = getattr(cell.font, 'color', None)
        if color is not None and getattr(color, 'type', None) == 'rgb':
            rgb = getattr(color, 'rgb', None)
            if isinstance(rgb, str) and len(rgb) >= 6:
                # Slice the last 6 characters to ignore Alpha channel (ARGB -> RGB)
                styles.append(f'color:#{rgb[-6:]};')

        # Extract background color
        fill = getattr(cell, 'fill', None)
        fgColor = getattr(fill, 'fgColor', None)
        if fgColor is not None and getattr(fgColor, 'type', None) == 'rgb':
            rgb = getattr(fgColor, 'rgb', None)
            # Filter out transparent/invalid colors (00000000 usually means no fill in some contexts, but checking length is safer)
            if isinstance(rgb, str) and rgb != '00000000' and len(rgb) >= 6:
                styles.append(f'background-color:#{rgb[-6:]};')

        return ''.join(styles)

    # Step 1: Validate file existence and format
    if not os.path.exists(file_path):
        return {"error": f"Error: File '{file_path}' does not exist.", "html_content": "", "file_metadata": {}}

    supported_formats = ['.xlsx', '.xls', '.csv']
    file_ext = os.path.splitext(file_path)[1].lower()

    if file_ext not in supported_formats:
        return {"error": f"Error: Unsupported file format '{file_ext}'.", "html_content": "", "file_metadata": {}}

    html_output = ""
    metadata = {
        "file_type": file_ext,
        "sheet_names": []
    }

    try:
```

```
        # Step 2: Handle CSV files
        if file_ext == '.csv':
            df = pd.read_csv(file_path)
            metadata["sheet_names"] = ["csv_data"]

            html_output += f"<h2>CSV : {os.path.basename(file_path)}</h2>\n"
            html_output += f"<p>Rows: {df.shape[0]}, Columns: {df.shape[1]}</p>\n"
            html_output += "<table border='1'>\n"

            # Add header
            html_output += "<tr>"
            for col in df.columns:
                html_output += f"<th>{col}</th>"
            html_output += "</tr>\n"

            # Add data rows
            for i, row in df.head(row_limit).iterrows():
                html_output += "<tr>"
                for value in row:
                    val_str = str(value) if pd.notna(value) else ""
                    html_output += f"<td>{val_str}</td>"
                html_output += "</tr>\n"

            if len(df) > row_limit:
                html_output += f"<tr><td colspan='{len(df.columns)}'>... ({len(df) - row_limit} more rows)</td></tr>\n"

            html_output += "</table>\n"

        # Step 3: Handle Excel files
        else:
            # data_only=True is essential to get values instead of formulas
            wb = load_workbook(file_path, data_only=True)
            metadata["sheet_names"] = wb.sheetnames

            html_output += f"<h1>Excel: {os.path.basename(file_path)}</h1>\n"

            # Step 4: Iterate through sheets
            for sheet in wb.worksheets:
                html_output += f"<h2>Sheet: {sheet.title}</h2>\n"

                max_row = sheet.max_row
                max_col = sheet.max_column

                html_output += f"<p>Rows: {max_row}, Columns: {max_col}</p>\n"
                html_output += "<table border='1' style='border-collapse:collapse;'>\n"

                # Step 5: Process rows and cells
                # enumerate(..., 1) makes i start at 1
                for i, row in enumerate(sheet.iter_rows(max_row=min(max_row, row_limit)), 1):
                    html_output += "<tr>"
                    for cell in row:
                        tag = "th" if i == 1 else "td"  # Assume first row is header

                        # Step 6: Extract style
                        style = get_cell_style(cell)
                        value = cell.value if cell.value is not None else ""

                        # Step 8: Construct HTML with inline styles
                        if style:
                            html_output += f"<{tag} style='{style}'>{value}</{tag}>"
```

Code 4: Tool Implenmenation for Parse Excel With Styles

```
import os
import pandas as pd

```

```python
def read_excel_basic(file_path: str, preview_rows: int = 50):
    """
    Basic Excel reader using standard Pandas functionality.
    Fails to capture style information required for color-based riddles.
    """

    # Step 1: Validate file existence
    if not os.path.exists(file_path):
        return {"error": f"File '{file_path}' not found."}

    file_ext = os.path.splitext(file_path)[1].lower()
    data_output = {}

    try:
        # Step 2: Read file based on extension
        # Pandas read_excel defaults to reading values
        if file_ext == '.csv':
            df = pd.read_csv(file_path)
            # Convert to markdown-style string for readability
            data_output['csv_data'] = df.head(preview_rows).to_markdown(index=False)

        elif file_ext in ['.xlsx', '.xls']:
            # sheet_name=None reads all sheets into a dictionary
            sheets = pd.read_excel(file_path, sheet_name=None)

            for sheet_name, df in sheets.items():
                # Replace NaNs with empty strings for cleaner looking tables
                df_clean = df.fillna("")

                # We limit the rows to avoid overwhelming the context window
                preview_df = df_clean.head(preview_rows)

                data_output[sheet_name] = preview_df.to_markdown(index=False)
        else:
            return {"error": "Unsupported file format."}

        # Step 4: Return result
        return {
            "file_name": os.path.basename(file_path),
            "sheet_content": data_output,
            "note": "Visual styles (colors, fonts) were not extracted."
        }

    except Exception as e:
        return {"error": f"Failed to parse file: {str(e)}"}

```

Code 5: Code Implenmenation for Read Excel Basic