ToolAtlas: Learning Once, Reusing Everywhere with Tool-Side Memory

Anonymous ACL submission

Abstract

Large language model (LLM) agents increasingly rely on external tools served by shared providers and accessed by heterogeneous downstream agents. Existing approaches improve tool use on the agent side through parameter updates, prompt refinement, or agent-side memory, making tool knowledge difficult to share and limited to behaviors observed in past tasks. We argue that reusable tool knowledge should instead be maintained by the tool provider. We introduce ToolAtlas, a graph-based framework that builds a persistent provider-side tool memory of tool capabilities, failure boundaries, and cross-tool compositions through execution-verified probing. At inference time, agents query the tool memory via adaptive graph traversal. Across two MCP-based benchmarks spanning eight services, ToolAtlas outperforms existing tool-side optimization and agent-side memory baselines by up to 21.61% in pass@1 and 18.61% in pass@4. The same tool memory also transfers across environment instances and agent frameworks without retraining or task-time exploration, yielding up to 24.16%/16.22% and 17.49%/14.27% relative gains in pass@1/pass@4, respectively. Ablation studies show that these gains arise from combining tool-centered memory organization with capability-guided execution probing. These results establish provider-side tool memory as an effective and reusable paradigm for tool servers. Our code is in: https://anonymous.4open.science/r/ToolAtlas-FE5E.

1 Introduction

Tool use has become a defining capability of modern large language model (LLM) agents, enabling them to invoke external services and data sources beyond their parametric knowledge. Recent work shows that such tool use now spans large, realistic ecosystems, from comprehensive tool-augmented benchmarks to thousands of real-world APIs (Li et al., 2023; Patil et al., 2024; Qin et al., 2024). Under the Model Context Protocol (MCP), this ecosystem is increasingly deployed in a server-centric form: tool providers expose capabilities through MCP servers, while heterogeneous agents consume them across products and environments (Model Context Protocol, 2026; OpenAI, 2026). Official provider-hosted servers from GitHub and Stripe further reflect this trend (GitHub, 2026; Stripe, 2026). In this setting, the same tool server may be called by many downstream agents with different prompts, action interfaces, and runtime environments.

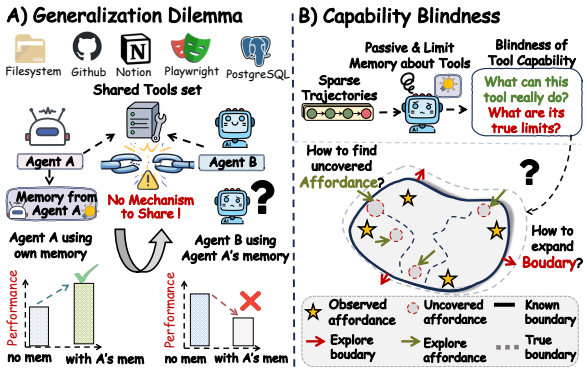

Figure 1: Two structural limitations of agent-side tool memory in shared tool server settings.

Despite this shared-server setting, most prior work still improves tool use on the agent side. Agent-side methods internalize tool-use behavior into model parameters (Schick et al., 2023; Qin et al., 2024; Liu et al., 2025b), create reusable skills or tools (Wang et al., 2024a; Cai et al., 2024; Qian et al., 2023; Yuan et al., 2024), or accumulate experience in agent-side memory (Zhao et al., 2024; Ouyang et al., 2025; Xu et al., 2025; Zhang et al., 2025a; Xiao et al., 2025). Existing tool-side efforts refine tool descriptions, documentation, retrieval, or catalog organization (Yuan et al., 2025; Fang et al., 2025; Wu et al., 2025a; Qu et al., 2025; Liu et al., 2025a). However, existing tool-side optimization mainly improves how tools are presented

or selected, while agent-side memory leaves knowledge inside the agent that produced it, rather than making it reusable across agents that use same tool.

This ownership mismatch matters in shared-server deployments. When many agents access one MCP server, each agent must rediscover behaviors and failure modes already observed by others. From the provider's perspective, this repeated exploration is wasteful: a single offline effort could be amortized across downstream agents if the resulting knowledge traveled with the tool. Moreover, passively logging agent trajectories reveals only the subset of tool behavior exercised by past tasks. These observations lead to two challenges:

- ***Generalization Dilemma***. Agent-side memory offers no mechanism for sharing what one agent learns with another. Each entry is written in the producing agent's own workflow, action format, reasoning style, and environment assumptions, so the same lesson must be relearned by every agent that calls the tool, even when the underlying tool behavior is unchanged (Figure 1(A)). Tool knowledge is therefore entangled with agent identity and difficult to reuse across agents.
- ***Capability Blindness***. Agent-side memory records only the tool behaviors exercised by past tasks. It follows the task distribution rather than the tool's capability, so untriggered boundary conditions and unused tool compositions are absent by construction (Figure 1(B)). The result is a sparse task footprint rather than a deliberate account of what the tool can do, where it fails, and how it composes with others.

These challenges suggest a two-part answer. Tool memory should be provider-side rather than agent-side, and it should be built through deliberate exploration of each tool's capability frontier rather than passive logging of tasks. To this end, we propose `ToolAtlas`, which maintains a provider-side tool memory graph for each tool server. The graph records what a tool can do, where it fails, and how it composes with peers, written once for any agent to consult. Because the graph is maintained with the tool, its construction cost can be amortized across many downstream agents. The graph has three interlinked layers. A Tool-Trace Graph distills past task executions into agent-neutral tool-rationale traces. A Tool-Capability Graph attaches affordance and boundary entries to each tool and connects tools through co-usage edges. A Tool-Strategy Graph extracts recurring orchestration principles from many tasks. To extend the graph along each tool's frontier rather than along incidental task coverage, `ToolAtlas` replaces passive trajectory collection with Frontier Exploration, an iterative loop in which an LLM probes the graph's current gaps along two complementary directions, outward toward untested boundary conditions and inward toward unverified affordances and tool compositions, executes each probe against the live tool, and folds the verified outcome back into the graph to sharpen next round. At inference, a lightweight navigator traverses the graph adaptively, expanding relevant regions step by step instead of retrieving a static top-k slice. Our contributions are threefold:

- We argue that tool memory should live with the tool rather than inside individual agents, so reusable tool knowledge is no longer tied to one agent's workflow or action format.
- We introduce `ToolAtlas`, a provider-side tool-memory framework that organizes knowledge as a tool memory graph, expands it through capability exploration, and retrieves it through adaptive graph traversal, making tool experience reusable across agents and less tied to past task coverage.
- Experiments on MCPMark and MCP-Universe across eight MCP services show that `ToolAtlas` outperforms tool-side optimization and agent-side memory baselines, while transferring across agents and task categories without re-training or task-time exploration.

2 Related Work

Agent-Side Tool-Use Improvement. Most prior work improves tool use on the agent side. One line internalizes tool-use behavior into model parameters through pre-training, supervised fine-tuning, or reinforcement learning (Schick et al., 2023; Qin et al., 2024; Liu et al., 2025b). Another line lets agents expand their own action space by creating reusable tools, programs, or skills, as in Voyager (Wang et al., 2024a) and tool-creation frameworks (Cai et al., 2024; Qian et al., 2023; Yuan et al., 2024). Closest to our setting are agent-side memory methods. ExpeL (Zhao et al., 2024) distills past trials into natural-language insights for later tasks. A-MEM (Xu et al., 2025) organizes memories as dynamically linked notes. ACE (Zhang et al., 2025a) treats context as an evolving playbook; and ToolMem learns tool-capability memories for multimodal agents (Xiao et al., 2025). These methods improve agent performance, but the acquired knowledge remains tied to

a particular model, skill library, or agent trajectory history, making it difficult to share across heterogeneous agents that access the same tool server.

Tool-Side Efforts and Interface Optimization. A complementary line improves how tools are exposed, selected, or organized for LLM agents. Tool documentation can serve as an effective zero-shot interface (Hsieh et al., 2023), motivating methods that optimize tool descriptions and usage examples, including EasyTool (Yuan et al., 2025), DRAFT (Qu et al., 2025), Play2Prompt (Fang et al., 2025), and ToolOptimal (Wu et al., 2025a). Other work handles large tool catalogs by retrieving, filtering, or structuring candidates, as in Re-Invoke (Chen et al., 2024), ToolRet (Shi et al., 2025), AnyTool (Du et al., 2024), ToolScope (Liu et al., 2025a), and ToolExpNet (Zhang et al., 2025b). These efforts improve tool interfaces or selection, but do not maintain provider-side memory of execution-verified capabilities, failure boundaries, and cross-tool compositions. `ToolAtlas` fills this gap with reusable tool-side memory built through frontier-guided execution probing and shared across heterogeneous agents without retraining or per-agent memory accumulation.

3 Methodology

`ToolAtlas` constructs a tool-side memory graph through three stages, as shown in Figure 2. First, *Memory Bootstrapping* uses tool specifications and web-retrieved usage snippets to generate seed tasks for every tool, executes the tasks with verifiers, and induces an initial graph from the resulting rollouts. Second, *Capability Exploration* uses the current graph to address Capability Blindness (Section 1) in two directions. It pushes outward to test tool capability boundaries and fills inward to uncover under-recorded affordances, meaning valid ways to use a tool that the graph has not yet captured, together with tool compositions. The generated probes are executed and their verified outcomes are written back to the graph. Third, at inference time, *Dynamic Memory Traversal* navigates the tool memory graph to gather related traces, tool entries, and strategies, then compresses the gathered information into task-conditioned guidance for the user agent. Appendix F shows a sample case study.

3.1 Memory Bootstrapping

Memory Bootstrapping gives a bare MCP service an initial tool-side memory. Let $\mathcal{T} = \{t_1, \ldots, t_M\}$ denote the tools exposed by the service, and let $\mathcal{M}_{\mathcal{T}}$ denote the tool memory graph built for them. For each tool $t_i \in \mathcal{T}$, `ToolAtlas` synthesizes k seed task-verifier pairs, defined as $\mathcal{D}_i^{(0)} = \{(Q_{ij}, v_{ij})\}_{j=1}^{k}$, where Q_{ij} is the j-th seed task for t_i and v_{ij} is its executable verifier. The full seed set is $\mathcal{D}^{(0)} = \bigcup_{i=1}^{M} \mathcal{D}_i^{(0)}$. The generated tasks are intended to cover representative uses of each tool. Since tool specifications are often too short to support diverse task synthesis, the task proposer also receives real-world usage snippets retrieved through web search. Each seed task Q_{ij} is executed for N independent rollouts, giving a rollout set $\mathcal{R}_{ij}^{(0)} = \{(\xi_{ij}^{(n)}, y_{ij}^{(n)})\}_{n=1}^{N}$, where $\xi_{ij}^{(n)}$ is the raw trajectory and $y_{ij}^{(n)} \in \{\textsc{Resolved}, \textsc{Failed}\}$ is the verifier-labeled outcome. These verified rollouts are then passed to Memory Induction.

Memory Induction. Memory Induction creates the tool memory graph from executed rollouts. During Memory Bootstrapping, it starts from an empty graph and incrementally inserts entries induced from each seed task and its rollout set. The motivation is that raw rollouts are still agent-specific because they contain the action syntax, retry pattern, and incidental reasoning style of the agent that produced them. Even rollouts for the same task may reveal different successful plans, repeated mistakes, or failure modes. Memory Induction therefore converts these rollouts into graph entries that keep reusable tool-use knowledge while discarding agent-specific form.

Tool-Trace Graph. Memory Induction condenses multiple rollouts of the same task into an *agent-neutral trace* $\tilde{\xi}_{ij}$, a sequence of *(tool, rationale)* pairs that removes agent-specific action syntax while preserving the tool-use intent. Together with the task and task-level tips $\mathcal{P}_{ij}$ distilled from successful and failed rollouts, this forms a trace node $q_{ij} = (Q_{ij}, \tilde{\xi}_{ij}, \mathcal{P}_{ij})$. The trace nodes form the Tool-Trace Graph $\mathcal{G}_{\mathsf{trace}} = (\mathcal{V}_{\mathsf{trace}}, \mathcal{E}_{\mathsf{trace}})$, where $\mathcal{V}_{\mathsf{trace}}$ is the set of trace nodes and semantic edges in $\mathcal{E}_{\mathsf{trace}}$ connect related trace nodes by embedding similarity over task summaries. For example, noisy rollouts for `list_directory_with_sizes` may contain dozens of calls and repeated replanning loops. Memory Induction compresses them into a short trace that keeps only the transferable tool-rationale sequence, such as discovering the workspace, inspecting candidate folders, checking sizes, verifying totals, and saving the final report.

Tool-Capability Graph. Agent-neutral traces make

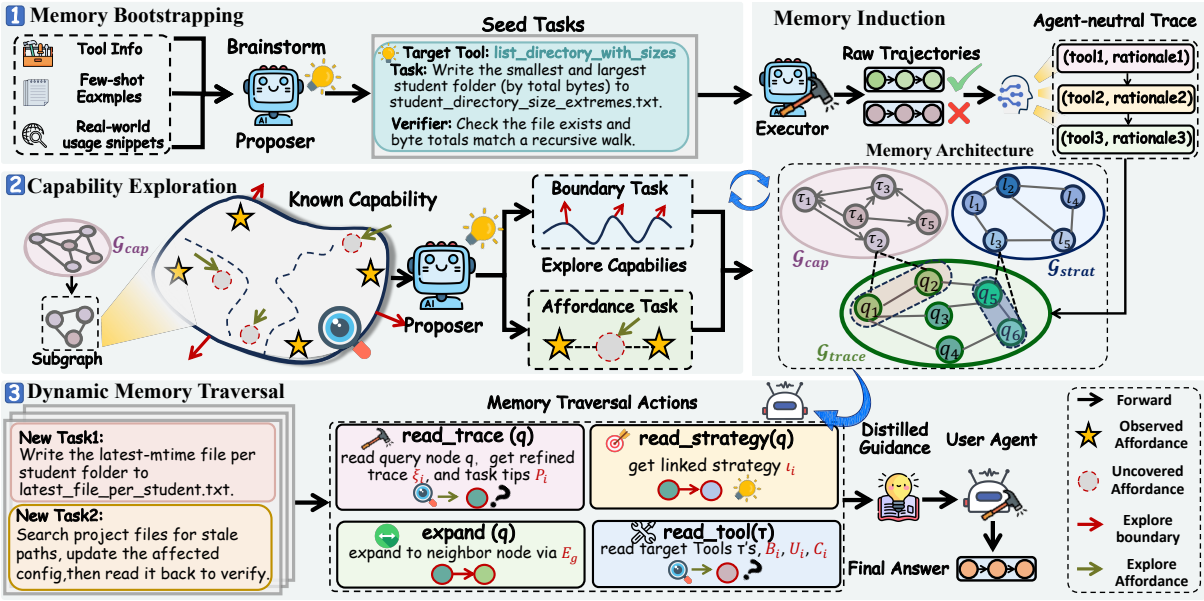

Figure 2: Overview of `ToolAtlas`. Stage 1 bootstraps a tool-side memory graph from seed tasks and verified rollouts, where Memory Induction converts raw trajectories into tool memory graph. Stage 2 uses the current graph to propose capability explorations that test tool boundaries and affordances, then writes verified outcomes back to the graph. Stage 3 traverses the graph at inference time and summarizes the retrieved traces, tool entries, and strategies into task-conditioned guidance for a user agent.

rollouts reusable, but they are still organized by task. To support tool-side capability discovery, `ToolAtlas` builds a Tool-Capability Graph $\mathcal{G}_{\text{cap}} = (\mathcal{V}_{\text{tool}}, \mathcal{E}_{\text{tool}})$ whose nodes are tools and whose inter-tool edges summarize recurrent co-usage patterns observed in agent-neutral traces. A tool node is connected to every trace node whose agent-neutral trace uses that tool, and we denote these trace-tool links by $\mathcal{E}_{\text{trace}\leftrightarrow\text{tool}}$. These links let the tool node gather what related trace nodes reveal about the tool. Each tool node t_i carries three entry sets $(\mathcal{U}_i, \mathcal{B}_i, \mathcal{C}_i)$, where affordance entries $\mathcal{U}_i$ record valid uses that t_i reliably supports, boundary entries $\mathcal{B}_i$ mark regimes where t_i becomes unreliable or infeasible, and co-usage entries $\mathcal{C}_i$ document recurrent compositions with peer tools. This graph is the operational state used later for capability exploration. It tells the proposer what is already known about a tool and which affordances, boundaries, or compositions remain under-covered.

Tool-Strategy Graph. Some lessons are planning patterns rather than single-tool properties. Memory Induction captures them by grouping semantically related trace nodes or trace nodes with shared tool-use patterns, then distilling recurring task tips into strategy nodes. These nodes form the Tool-Strategy Graph $\mathcal{G}_{\text{strat}} = (\mathcal{V}_{\text{strat}}, \mathcal{E}_{\text{strat}})$. We denote bidirectional trace-strategy links by $\mathcal{E}_{\text{trace}\leftrightarrow\text{strat}}$. For example, file-system traces that first check allowed directories and locate the writable workspace can induce a strategy to discover the workspace and target scope before acting. This graph lets a user agent reuse high-level plans on similar tasks.

After Memory Induction, the complete tool memory graph consists of the three graphs, $\mathcal{M}_{\mathcal{T}} = (\mathcal{G}_{\text{trace}}, \mathcal{G}_{\text{cap}}, \mathcal{G}_{\text{strat}})$ with cross-graph links $\mathcal{E}_{\text{trace}\leftrightarrow\text{tool}}$ between trace nodes and supported tool nodes, and $\mathcal{E}_{\text{trace}\leftrightarrow\text{strat}}$ between trace nodes and supported strategy nodes.

3.2 Capability Exploration

Memory Bootstrapping gives the MCP service an initial tool-side memory, but seed tasks still cover only a small region of the tool capability. Capability Exploration expands $\mathcal{M}_{\mathcal{T}}$ by using the Tool-Capability graph as feedback. The graph records what is already known about each tool, and the task proposer uses that state to propose tasks that test what remains uncertain. At round r, `ToolAtlas` samples a local subgraph $\mathcal{S}^{(r)}$ of K related tools from the current Tool-Capability Graph $\mathcal{G}_{\text{cap}}^{(r-1)}$. The sampler starts from a target tool and follows co-usage edges, favoring plausible compositions while preserving coverage of under-explored tools.

For each target tool $t_i \in \mathcal{S}^{(r)}$, `ToolAtlas` builds a proposer context $\mathcal{K}_i^{(r)}$ from the target

tool node, its entries $(\mathcal{U}_i, \mathcal{B}_i, \mathcal{C}_i)$, neighboring tools, and linked traces. Conditioned on this context, an LLM proposer generates probe tasks $\mathcal{D}_{i,d}^{(r)}$ along two directions, $d \in \{\text{boundary}, \text{affordance}\}$. The outward direction tests capability boundaries where a tool becomes unreliable or infeasible, such as asking `list_directory_with_sizes` to rank deeply nested folders by recursive byte total when its node records shallow per-entry sizes. The inward direction uncovers under-recorded affordances and compositions, such as testing whether `list_directory_with_sizes` with `get_file_info` can find the largest file under each student directory. Each proposed task is paired with a verifier and executed by the agent. The resulting probe rollouts $\mathcal{R}^{(r)}$ are passed through Memory Induction to update $\mathcal{M}_{\mathcal{T}}^{(r-1)}$ into $\mathcal{M}_{\mathcal{T}}^{(r)}$, adding trace nodes, tool capability entries, and strategy nodes.

3.3 Dynamic Memory Traversal

At inference time, **ToolAtlas** guides a user agent on a new task by walking the tool memory graph. Fixed top-k retrieval is insufficient because evidence may span trace, tool, and strategy graph layers. Dynamic Memory Traversal performs a graph walk over $\mathcal{M}_{\mathcal{T}}$. Given a new task Q, traversal retrieves the k_r nearest Tool-Trace nodes as the initial context C_0. At step s, a lightweight navigator selects graph navigation operation a_s updates the context as $C_{s+1} = C_s \cup \textsc{Navigate}(a_s; \mathcal{M}_{\mathcal{T}})$. The navigator uses four operations: (1) $\textsc{ReadTrace}(q)$ reads the task, agent-neutral trace, and tips of trace node q; (2) $\textsc{Expand}(q)$ follows trace-graph edges to similar trace nodes; (3) $\textsc{ReadTool}(t)$ reads affordance, boundary, and co-usage entries from tool node t; and (4) $\textsc{ReadStrategy}(q)$ reads strategy nodes supported by trace node q. Traversal stops when the navigator judges the context sufficient or the read budget is exhausted. The final context is compressed into task-conditioned guidance $g(Q)$ and injected into the user agent's memory slot before rollout.

4 Experiment

Our experiments evaluate the effectiveness and generalization of **ToolAtlas**. Specifically, we ask whether the provider-side tool memory can be built once, and then reused by different agents, models, and environment instances. We organize the evaluation around four questions: (**RQ1**) How does **ToolAtlas** perform compared with tool-side optimization and agent-side memory baselines? (**RQ2**) Does the tool-side memory constructed by **ToolAtlas** generalize across environment instances and agent frameworks? (**RQ3**) How much does each key design choice in **ToolAtlas** contribute to its performance gains? (**RQ4**) What inference cost does **ToolAtlas** incur, and how does its utility-cost trade-off compare with baselines?

4.1 Experimental Setup

Benchmarks. To thoroughly examine the effectiveness of **ToolAtlas**, We evaluate on two MCP-based benchmarks covering eight services. First, we use MCPMARK (Wu et al., 2025b), which includes *Filesystem*, *GitHub*, *Notion*, *Playwright*, and *PostgreSQL*. Second, we use MCP-UNIVERSE (Luo et al., 2025), from which we select *Location Navigation*, *Financial Analysis*, and *Web Searching*. We exclude MCP-UNIVERSE domains that overlap with MCPMARK so that the combined evaluation covers a broader range of tool interfaces and task types. Dataset details and split construction are provided in Appendix A.1.

Baselines. We compare **ToolAtlas** against two complementary families of methods. The first targets *tool-side optimization*, which improve the descriptions, examples, or instructions of the tools, including EasyTool (Yuan et al., 2025), Play2Prompt (Fang et al., 2025), and ToolOptimal (Wu et al., 2025a). The second family are *agent-side memory* methods, which summarize past trajectories into reusable experience associated with a particular agent, including ExpeL (Zhao et al., 2024), A-MEM (Xu et al., 2025), and ACE (Zhang et al., 2025a). We also include *Vanilla*, the agent with raw MCP tool specifications and no augmentation. Full details are in Appendix A.2.

Agents and LLM Backbones. We instantiate three representative tool-using agent frameworks (details in Appendix A.3). **ReAct** (Yao et al., 2023) issues each action as a single function call. **CodeAct** (Wang et al., 2024b) issues actions as executable Python snippets. **SwitchAct** is a hybrid agent that we implement, exposing both interfaces to the LLM and allowing it to choose at each step. These frameworks cover both function-calling and code-based tool-use paradigms. Unless otherwise stated, we use SwitchAct as the default inference agent. We evaluate three LLM backbones from two model families: GPT-5.4, GPT-5.4-mini, and Grok-4 (implementation details in Appendix A.4).

Method	Filesystem		GitHub		Notion		Playwright		PostgreSQL		Overall	
	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑
GPT-5.4												
Vanilla	48.68	68.42	46.67	53.33	44.44	55.56	46.15	61.54	59.62	69.23	49.11	61.62
EasyTool	52.63	78.95	46.67	66.67	45.83	66.67	50.00	61.54	59.62	76.92	50.95	70.15
Play2Prompt	50.00	78.95	51.67	66.67	48.61	61.11	53.85	69.23	61.54	76.92	53.13	70.58
ToolOptimal	53.95	84.21	48.33	60.00	48.61	66.67	50.00	61.54	53.85	69.23	50.95	68.33
Expel	48.68	78.95	36.67	53.33	44.44	66.67	55.77	69.23	55.77	76.92	48.27	69.02
A-Mem	57.89	78.95	46.67	60.00	43.06	55.56	42.31	53.85	59.62	**84.61**	49.91	66.59
ACE	59.21	84.21	53.33	66.67	50.00	66.67	53.85	61.54	65.38	**84.61**	56.35	72.74
`ToolAtlas`	**69.73**	**94.73**	**56.67**	**73.33**	**51.39**	**72.22**	**59.62**	**76.92**	**67.30**	**84.61**	**60.94**	**80.36**
Impro(%)	*+17.77*	*+12.49*	*+6.26*	*+9.99*	*+2.78*	*+8.32*	*+6.90*	*+11.11*	*+2.94*	*+0.00*	*+8.15*	*+10.48*
GPT-5.4-mini												
Vanilla	15.79	26.32	15.00	33.33	26.39	44.44	15.38	38.46	46.15	61.54	23.74	40.82
EasyTool	22.37	52.63	26.67	46.67	23.61	38.89	21.15	61.54	50.00	61.54	28.76	52.25
Play2Prompt	19.74	47.37	28.33	46.67	26.39	**55.56**	17.31	53.85	50.00	69.23	28.35	54.54
ToolOptimal	23.68	52.63	28.33	40.00	22.22	50.00	15.38	38.46	53.85	69.23	28.69	50.06
Expel	17.11	42.11	26.67	46.67	26.39	44.44	26.92	46.15	48.08	61.54	29.03	48.18
A-Mem	21.05	52.63	28.33	40.00	23.61	44.44	23.08	46.15	53.85	61.54	29.98	48.95
ACE	23.68	52.63	28.33	40.00	31.94	44.44	32.69	53.85	53.85	69.23	34.10	52.03
`ToolAtlas`	**28.94**	**68.42**	**31.67**	**53.33**	**34.72**	**55.56**	**36.53**	**69.23**	**55.77**	**76.92**	**37.53**	**64.69**
Impro(%)	*+22.21*	*+30.00*	*+11.79*	*+14.27*	*+8.70*	*+0.00*	*+11.75*	*+12.50*	*+3.57*	*+11.11*	*+10.06*	*+18.61*

Table 1: Performance comparison on MCPMARK under the same-environment split, using *SwitchAct* with GPT-5.4 and GPT-5.4-mini as backbones. Results with Grok-4 are reported in Table 8. The **best** and second best results are marked in bold and with underline, respectively. *Impro(%)* is the relative gain of `ToolAtlas` over the strongest baseline.

Method	Filesystem		GitHub		Notion		Playwright		PostgreSQL		Overall	
	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑
Vanilla	48.53	58.82	44.44	66.67	36.76	58.82	42.86	57.14	55.77	61.54	45.67	60.60
EasyTool	51.47	70.59	44.44	55.56	20.59	41.18	53.57	64.29	57.69	69.23	45.55	60.17
Play2Prompt	52.94	**76.47**	47.22	66.67	25.00	47.06	50.00	64.29	55.77	61.54	46.19	63.21
ToolOptimal	41.18	64.71	30.56	55.56	27.94	52.94	44.64	**71.43**	55.77	69.23	40.02	62.77
Expel	45.59	70.59	30.56	44.44	29.41	47.06	39.29	57.14	48.08	61.54	38.59	56.15
A-Mem	44.23	53.85	36.11	44.44	32.35	52.94	35.71	**71.43**	46.15	61.54	38.91	56.84
ACE	44.12	58.82	38.89	55.56	30.88	41.18	35.71	42.86	51.92	69.23	40.30	53.53
`ToolAtlas`	**61.76**	**76.47**	**58.33**	**77.78**	**44.12**	**64.71**	**57.14**	**71.43**	**65.38**	**76.92**	**57.35**	**73.46**
Impro(%)	*+16.66*	*+0.00*	*+23.53*	*+16.66*	*+20.02*	*+10.01*	*+6.66*	*+0.00*	*+13.33*	*+11.11*	*+24.16*	*+16.22*

Table 2: Performance comparison on MCPMARK under the cross-environment split, using *SwitchAct* with GPT-5.4 as the backbone. The **best** and second best results are marked in bold and with underline, respectively. *Impro(%)* is the relative gain of `ToolAtlas` over the strongest baseline.

Metrics and Protocol. We report pass@1 and pass@4 following the benchmarks' setting, and total inference tokens to measure cost. For methods that construct memory, the memory is built only on the designated training split and then frozen before evaluation. For `ToolAtlas`, the resulting tool memory is associated with the tool rather than with a specific downstream agent. This allows us to evaluate whether the same tool-side memory can be reused across agents and environments without retraining or additional trajectory accumulation.

4.2 RQ1: Effectiveness of `ToolAtlas`

We first evaluate on the *same-environment split* (Appendix A.1), where training and test tasks share the same environment instances. This setting isolates the effect of the memory mechanism, since no environment-instance shift is introduced between memory construction and evaluation. Tables 1 and 8 report results on MCPMARK across three LLM backbones, and Table 9 reports results on MCP-UNIVERSE. `ToolAtlas` outperforms the strongest baseline on every service and with every backbone on MCPMARK. Relative to the best baseline, `ToolAtlas` improves pass@1/pass@4 by 8.15%/10.48% with GPT-5.4, 10.06%/18.61% with GPT-5.4-mini, and 21.61%/16.11% with Grok-4. The same trend extends to MCP-UNIVERSE results. These results show that the gains are not tied to a single benchmark, or model backbone. Instead, they support the central premise of `ToolAtlas`: execution-verified, tool-bound knowledge can improve downstream tool use even when the agent itself is unchanged.

Method	Filesystem		GitHub		Notion		Playwright		PostgreSQL		Overall	
	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑
Test on ReAct												
Vanilla	51.32	63.16	48.33	53.33	43.06	61.11	32.69	53.85	48.08	69.23	44.70	60.14
EasyTool	60.53	73.68	43.33	46.67	41.67	61.11	34.62	53.85	42.31	61.54	44.49	59.37
Play2Prompt	52.63	73.68	41.67	46.67	41.67	**66.67**	46.15	61.54	55.77	61.54	47.58	62.02
ToolOptimal	57.89	78.95	36.67	40.00	43.06	**66.67**	36.54	53.85	51.92	61.54	45.22	60.20
Expel	51.32	73.68	46.67	53.33	40.28	61.11	30.77	46.15	46.15	69.23	43.04	60.70
A-Mem	50.00	78.95	45.00	60.00	25.00	61.11	25.00	61.54	34.62	69.23	35.92	66.17
ACE	55.26	63.16	45.00	46.67	38.89	55.56	34.62	53.85	50.00	61.54	44.75	56.16
`ToolAtlas`	**64.47**	**84.21**	**50.00**	**73.33**	**48.61**	**66.67**	**51.92**	**69.23**	**61.54**	**84.62**	**55.31**	**75.61**
Impro(%)	*+6.51*	*+6.66*	*+3.46*	*+22.22*	*+12.89*	*+0.00*	*+12.50*	*+12.50*	*+10.35*	*+22.23*	*+16.25*	*+14.27*
Test on CodeAct												
Vanilla	53.95	68.42	21.67	40.00	41.67	61.11	42.31	53.85	57.69	76.92	43.46	60.06
EasyTool	65.79	84.21	26.67	33.33	43.06	66.67	44.23	53.85	61.54	69.23	48.26	61.46
Play2Prompt	53.95	78.95	33.33	46.67	51.39	66.67	42.31	53.85	57.69	76.92	47.73	64.61
ToolOptimal	64.47	84.21	28.33	33.33	45.83	61.11	40.38	53.85	53.85	61.54	46.57	58.81
Expel	56.58	68.42	20.00	33.33	40.28	55.56	40.38	46.15	57.69	69.23	42.99	54.54
A-Mem	56.58	84.21	21.67	26.67	43.06	55.56	46.15	**61.54**	30.77	61.54	39.65	57.90
ACE	52.63	63.16	18.33	40.00	41.67	61.11	40.38	53.85	55.77	61.54	41.76	55.93
`ToolAtlas`	**78.95**	**89.47**	**38.33**	**53.33**	**52.78**	**72.22**	**48.08**	**61.54**	**65.38**	**92.31**	**56.70**	**73.77**
Impro(%)	*+20.00*	*+6.25*	*+15.00*	*+14.27*	*+2.70*	*+8.32*	*+4.18*	*+0.00*	*+6.24*	*+20.01*	*+17.49*	*+14.18*

Table 3: Performance comparison on MCPMARK under the cross-agent transfer setting, using GPT-5.4 as the backbone. All baselines build their memory with *SwitchAct* as the source agent, then transfer to two target agents: *ReAct* and *CodeAct*. The **best** and second best results are marked in bold and with underline, respectively. *Impro(%)* is the relative gain of `ToolAtlas` over the strongest baseline.

Method	Filesystem		GitHub		Notion		Playwright		PostgreSQL		Overall	
	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑
`ToolAtlas` (Full)	**69.73**	**94.73**	**56.67**	**73.33**	**51.39**	**72.22**	**59.62**	**76.92**	**67.30**	**84.61**	**60.94**	**80.36**
Brainstorm Mechanism												
w/o Boundary	65.79	89.47	55.00	66.67	48.61	66.67	53.85	69.23	63.46	76.92	57.34	73.79
w/o Affordance	61.84	84.21	48.33	60.00	50.00	66.67	50.00	61.54	57.69	69.23	53.57	68.33
Memory Organization												
w/o $\mathcal{G}_{\text{cap}}$	55.26	78.95	50.00	60.00	45.83	61.11	51.92	61.54	55.77	69.23	51.76	66.17
w/o $\mathcal{G}_{\text{strat}}$	68.42	94.73	53.33	66.67	48.61	66.67	57.69	76.92	65.38	76.92	58.69	76.38
w/o Dynamic Traversal	67.11	89.47	51.67	66.67	50.00	66.67	55.77	69.23	61.54	76.92	57.22	73.79

Table 4: Ablation study on MCPMARK under the same-environment split, using *SwitchAct* with GPT-5.4 as the backbone. The **best** results are marked in bold.

The relative gains are larger for GPT-5.4-mini and Grok-4 than for GPT-5.4, which suggests that a provider-side tool memory does more than improve already strong agents. It can also compensate for weaker backbones by supplying reusable knowledge about tool behavior at inference time.

4.3 RQ2: Generalization of `ToolAtlas`

The defining advantage of `ToolAtlas` is not only that it improves one agent, but that it places tool-use knowledge at the provider side, where the same tool memory can serve many downstream consumers. We therefore evaluate its generalization under two deployment shifts: cross-environment transfer, where the tool service is the same but the underlying environment instance changes, and cross-agent transfer, where the tool memory is built with one agent and consumed by others.

Across Environment Instances. In this setting, training and test tasks use disjoint environment instances while keeping the tool set fixed (Appendix A.1). For example, the tool memory may be built from one repository, workspace, or database instance, and evaluated on another. This setting tests whether a method learns generalized tool behaviors rather than memorizing environment-specific artifacts. Table 2 shows that `ToolAtlas` achieves the best performance and outperforms the strongest baseline in every service, with relative gains of 24.16%/16.22% on pass@1/pass@4. In contrast, the same shift degrades many tool-side and agent-side baselines that fall 5–7 percentage points below *Vanilla* on overall pass@1. This indicates that these methods are absorbing environment-specific patterns that cannot transfer. By organizing memory around tool capabilities and

validating entries through execution, `ToolAtlas` captures knowledge that remains useful across environment instances of the same service.

Across Agent Frameworks. We test whether different agent frameworks can reuse the tool memory. Each method first constructs memory with SwitchAct and then transfers it to ReAct, which uses function calls, and CodeAct, which uses executable Python snippets, respectively. This setting reflects a realistic deployment: the provider maintains one tool-side memory, while heterogeneous downstream agents query the same memory instance. `ToolAtlas` achieves the largest gains on both target agents as shown in Table 3: $+16.25\%/+14.27\%$ on ReAct and $+17.49\%/+14.18\%$ on CodeAct, measured by overall pass@1/pass@4. In contrast, agent-side memory baselines often fall below *Vanilla* after transfer. For example, A-MEM loses 8.78% on ReAct overall pass@1 and 26.92% percentage points on CodeAct PostgreSQL pass@1. These failures highlight a limitation of agent-bound memory: the stored lessons encode the workflow, action format, and failure modes of the specific agent that generated them, while `ToolAtlas` avoids this coupling by storing generalized knowledge at the tool level.

4.4 RQ3: Ablation Study

Table 4 presents an ablation of `ToolAtlas` by isolating each design choice in the brainstorm mechanism and the memory organization. Removing any component hurts performance, showing that the gains come from coupling a tool-centered memory structure with frontier-guided execution probing. The largest drop comes from removing the Tool-Capability Graph: *w/o* $\mathcal{G}_{\text{cap}}$ decreases pass@1/pass@4 by $9.18\%/14.19\%$. This directly supports the design principle of `ToolAtlas`: reusable tool knowledge should be centered at the tool and its capabilities, not at the idiosyncratic trajectory of the agent that discovered it. During the brainstorm phase, removing affordance exploration causes larger degradation than removing boundary exploration. Specifically, *w/o Affordance* reduces pass@1/pass@4 by $7.37\%/12.03\%$, whereas *w/o Boundary* reduces by $3.60\%/6.57\%$. This suggests that affordance exploration is the primary source of positive tool-use knowledge, while boundary exploration provides complementary information about where tools fail or behave unreliably. Removing *Dynamic Traversal* and the *Tool-Strategy Graph* also produces consistent drops, showing that adaptive exploration and higher-level organization further improve the atlas beyond the basic tool-bound memory structure. Appendix C.3 further shows that Frontier Exploration avoids the plateau of same-budget seed-task sampling, suggesting `ToolAtlas` constructs less redundant tool memory.

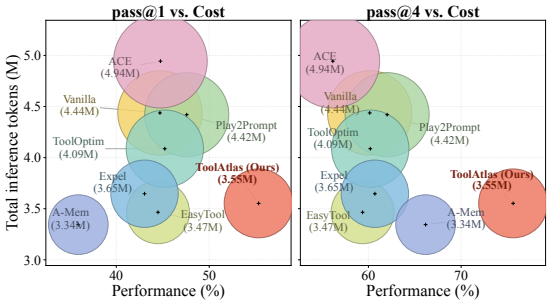

Figure 3: Utility–cost trade-off on MCPMARK under the cross-agent transfer setting, using GPT-5.4 as the backbone. Total inference tokens are annotated per method.

4.5 RQ4 Inference and Utility Cost Trade-off

Under the cross-agent transfer setting (Section 4.3), Figure 3 plots pass@1/pass@4 against inference tokens on MCPMARK. `ToolAtlas` lies on the Pareto frontier in both, reaching highest task success with 3.55M tokens, below Vanilla's 4.44M. The saving comes from avoiding tool rediscovery, since provider-side memory supplies execution-verified affordances and boundaries upfront, so rollout savings exceed the prompt-side cost. Baselines miss this balance. Tool-side optimization is compact but less accurate, while agent-side memory adds prompt tokens without matching gains. `ToolAtlas` avoids both ends by storing execution-verified, agent-neutral tool knowledge with the tool, giving the receiving agent a generalizable capability map instead of forcing re-exploration.

5 Conclusion

Tool memory should be maintained provider-side rather than inside individual agents. We realize this view in `ToolAtlas`, which charts each MCP server as a three-layer tool memory graph, expands it through brainstorm-driven exploration of tool affordances and boundaries, and surfaces task-conditioned guidance at inference. Across eight services and two benchmarks, `ToolAtlas` outperforms tool-side and agent-side baselines, transfers across environment instances and agent frameworks without retraining, and reduces end-to-end inference cost. Provider-side memory offers a practical paradigm for shared tool servers, amortizing one offline effort across many downstream agents.

6 Limitations

Although ToolAtlas demonstrates strong effectiveness and transferability on MCP-based benchmarks, several limitations remain. Our evaluation focuses on eight services from two MCP benchmarks and three LLM backbones. This setting covers diverse tool types, but it does not exhaust the full space of real-world tool servers. In particular, we do not evaluate tools in highly specialized domain constraints or rapidly changing APIs. Extending the evaluation to broader production tool ecosystems is an important direction for future work. In addition, our study emphasizes task success, transfer across environment instances and agent frameworks, component ablations, and inference cost. We do not fully account for the total lifecycle of provider-side tool-memory, especially regarding to re-verification, storage, and governance. Tool behavior may change as APIs are updated, permissions are modified, or backend data schemas evolve. A practical deployment of ToolAtlas would therefore require dedicated verification and refresh mechanisms to ensure that stored tool knowledge remains accurate over time.

7 Ethical Considerations

All experiments in this study were conducted on publicly available MCP-based benchmarks, including MCPMark and MCP-Universe, in compliance with their respective licenses and usage terms. When external services or accounts were required to instantiate benchmark environments, we used researcher-controlled accounts and synthetic or benchmark-provided data only. No third-party user accounts, private user data, or production workloads were involved. This work did not involve human or animal subjects, and no personally identifiable information was collected, or used. We further note that a provider-side tool memory could, in principle, be poisoned or exploited if constructed from untrusted execution traces, and any deployment should pair it with verification and access-control mechanisms before serving downstream agents.

References

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, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Yanfei Chen, Jinsung Yoon, Devendra Singh Sachan, Qingze Wang, Vincent Cohen-Addad, Mohammad-Hossein Bateni, Chen-Yu Lee, and Tomas Pfister. 2024. Re-invoke: Tool invocation rewriting for zero-shot tool retrieval. In *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*, Findings of ACL, pages 4705–4726. Association for Computational Linguistics.

Yu Du, Fangyun Wei, and Hongyang Zhang. 2024. Anytool: Self-reflective, hierarchical agents for large-scale API calls. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 11812–11829. PMLR / OpenReview.net.

Wei Fang, Yang Zhang, Kaizhi Qian, James R. Glass, and Yada Zhu. 2025. PLAY2PROMPT: zero-shot tool instruction optimization for LLM agents via tool play. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, Findings of ACL, pages 26274–26290. Association for Computational Linguistics.

GitHub. 2026. Using the github mcp server in your ide. https://docs.github.com/en/copilot/how-tos/provide-context/use-mcp-in-your-ide/use-the-github-mcp-server. Accessed: 05-01-2026.

Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. 2023. Tool documentation enables zero-shot tool-usage with large language models. *CoRR*, abs/2308.00675.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 3102–3116. Association for Computational Linguistics.

Marianne Menglin Liu, Daniel Garcia, Fjona Parllaku, Vikas Upadhyay, Syed Fahad Allam Shah, and Dan Roth. 2025a. Toolscope: Enhancing LLM agent tool use through tool merging and context-aware filtering. *CoRR*, abs/2510.20036.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong Wang, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Xinzhi Wang, Yong Liu, Yasheng Wang, and 8 others. 2025b. Toolace: Winning the points of LLM function calling. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo,

Silvio Savarese, Caiming Xiong, and Junnan Li. 2025. Mcp-universe: Benchmarking large language models with real-world model context protocol servers. *CoRR*, abs/2508.14704.

Model Context Protocol. 2026. What is the model context protocol (mcp)? https://modelcontextprotocol.io/docs/getting-started/intro. Accessed: 05-01-2026.

OpenAI. 2026. Mcp and connectors. https://developers.openai.com/api/docs/guides/tools-connectors-mcp. Accessed: 05-01-2026.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. 2025. Reasoningbank: Scaling agent self-evolving with reasoning memory. *CoRR*, abs/2509.25140.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: Large language model connected with massive apis. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Cheng Qian, Chi Han, Yi Ren Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. CREATOR: tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, Findings of ACL, pages 6922–6939. Association for Computational Linguistics.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2025. From exploration to mastery: Enabling llms to master tools via self-driven interactions. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools.

Zhengliang Shi, Yuhan Wang, Lingyong Yan, Pengjie Ren, Shuaiqiang Wang, Dawei Yin, and Zhaochun Ren. 2025. Retrieval models aren't tool-savvy: Benchmarking tool retrieval for large language models. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, Findings of ACL, pages 24497–24524. Association for Computational Linguistics.

Stripe. 2026. Model context protocol (mcp). https://docs.stripe.com/mcp. Accessed: 05-01-2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024a. Voyager: An open-ended embodied agent with large language models. *Trans. Mach. Learn. Res.*, 2024.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024b. Executable code actions elicit better LLM agents. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 50208–50232. PMLR / OpenReview.net.

Bin Wu, Edgar Meij, and Emine Yilmaz. 2025a. A joint optimization framework for enhancing efficiency of tool utilization in LLM agents. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, Findings of ACL, pages 22361–22373. Association for Computational Linguistics.

Zijian Wu, Xiangyan Liu, Xinyuan Zhang, Lingjun Chen, Fanqing Meng, Lingxiao Du, Yiran Zhao, Fanshi Zhang, Yaoqi Ye, Jiawei Wang, Zirui Wang, Jinjie Ni, Yufan Yang, Arvin Xu, and Michael Qizhe Shieh. 2025b. Mcpmark: A benchmark for stress-testing realistic and comprehensive MCP use. *CoRR*, abs/2509.24002.

Yunzhong Xiao, Yangmin Li, Hewei Wang, Yunlong Tang, and Zora Zhiruo Wang. 2025. Toolmem: Enhancing multimodal agents with learnable tool capability memory. *CoRR*, abs/2510.06664.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. A-MEM: agentic memory for LLM agents. *CoRR*, abs/2502.12110.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi Fung, Hao Peng, and Heng Ji. 2024. CRAFT: customizing llms by creating and retrieving from specialized toolsets. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Kan Ren, Dongsheng Li, and Deqing Yang. 2025. EASYTOOL: enhancing llm-based agents with concise tool instruction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational*

Linguistics: Human Language Technologies, NAACL 2025 - Volume 1: Long Papers, Albuquerque, New Mexico, USA, April 29 - May 4, 2025, pages 951–972. Association for Computational Linguistics.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. 2025a. Agentic context engineering: Evolving contexts for self-improving language models. *CoRR*, abs/2510.04618.

Zijing Zhang, Zhanpeng Chen, He Zhu, Ziyang Chen, Nan Du, and Xiaolong Li. 2025b. Toolexpnet: Optimizing multi-tool selection in llms with similarity and dependency-aware experience networks. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, Findings of ACL, pages 15706–15722. Association for Computational Linguistics.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. Expel: LLM agents are experiential learners. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 19632–19642. AAAI Press.

A Experimental Details

A.1 Dataset Descriptions

We evaluate `ToolAtlas` on two MCP-based benchmarks that test tool-using capabilities across diverse real-world environments. We first introduce the terminology used to describe benchmark organization:

- **MCP service** denotes a tool platform (*e.g.*, *Filesystem*, *GitHub*). The tool set of each service is fixed and shared across all its environment instances and tasks.
- **Environment instance** denotes one of multiple concrete, pre-populated realizations of an MCP service (*e.g.*, a specific repository, workspace, database, or directory tree), each reusable across multiple tasks.
- **Task** denotes a single evaluation unit defined over an environment instance, comprising a natural-language instruction, a programmatic verifier, and a task-specific starting snapshot of that instance.

We adopt two benchmarks built on real-world MCP servers:

- **MCPMark** (Wu et al., 2025b) (available at `https://mcpmark.ai/`) is a stress-testing benchmark comprising 121 tasks across five MCP services with curated environment instances. Each task pairs a natural-language instruction with a programmatic verifier and is executed from a task-specific starting snapshot. We adopt its five services as follows:
  - **Filesystem**: file I/O, directory organization, metadata inspection, and search over curated folder structures mirroring everyday user scenarios (*e.g.*, desktop files, legal documents, student databases).
  - **GitHub**: project management and Git operations including CI/CD, issues, branches, pull requests, and commits, grounded in repositories with realistic development histories.
  - **Notion**: creating, editing, and querying documents and databases via the official Notion API, instantiated from widely adopted templates (*e.g.*, online resume, travel planner, student dashboard).
  - **Playwright**: browser automation including navigation, form completion, data extraction, and screenshot generation.

- **PostgreSQL**: schema exploration and SQL query execution over representative template databases with realistic schemas (*e.g.*, Chinook, DVD Rental, Employees, Lego).

- **MCP-Universe** (Luo et al., 2025) (available at https://github.com/SalesforceAIResearch/MCP-Universe) is a benchmark of 231 tasks spanning six MCP service. We adopt three domains and exclude three (3D Design, Repository Management, and Browser Automation) that overlap with services already covered by MCPMARK. The three adopted MCP services are:
  - **Location Navigation**: geographic queries and route planning via the Google Maps MCP server, including distance optimization, stop finding, and multi-point navigation.
  - **Financial Analysis**: stock and financial data analysis via the YFinance MCP server, covering portfolio analysis, trading strategies, and institutional holdings.
  - **Web Search**: open-domain information retrieval using Google Search and Fetch MCP servers, spanning person identification, entity discovery, and complex reasoning.

MCP Service	#Tools	#Instances	Train	Test
Filesystem	14	10	11	19
GitHub	94	6	8	15
Notion	19	10	10	18
Playwright	21	3	6	13
PostgreSQL	9	7	8	13
Total	*157*	*36*	*43*	*78*

Table 5: Task distribution of MCPMARK under the same-environment split.

MCP Service	#Tools	Train	Test
Location Navigation	7	9	26
Financial Analysis	10	10	30
Web Search	2	13	37
Total	*19*	*32*	*93*

Table 6: Task distribution of the adopted MCP-UNIVERSE domains under the same-environment split.

Data Split Protocol. We split each benchmark for two purposes, corresponding to two generalization settings:

- **Same-environment split.** This is the most basic setting for memory accumulation: train and test tasks *share the same environment instances and the same tool set*, isolating the effect of the memory itself from any environment or tool shift. Within each MCP service, tasks are randomly partitioned into disjoint train and test sets at a ratio of roughly 1:2, while allowing both splits to draw from the same pool of environment instances. Each task still begins from its own task-specific snapshot, so train and test tasks never overlap at the task level. Statistics are summarized in Table 5 (MCPMARK) and Table 6 (MCP-UNIVERSE).
- **Cross-environment split.** This setting tests whether memory generalizes beyond the environment instances seen during construction: train and test tasks *share the same tool set but no longer share environment instances*. Within each MCP service, environment instances themselves are randomly partitioned into disjoint train and test groups at a ratio of roughly 2:3; all tasks associated with a train-side instance are used for memory construction, and all tasks associated with a test-side instance are used for evaluation. The resulting task counts therefore follow the instance assignment rather than a pre-specified task-level ratio. MCP-UNIVERSE does not define environment instances and is therefore excluded from this protocol. The instance-level assignment is summarized in Table 7.

Metrics. We report pass@1 and pass@4 as the primary evaluation metrics, following the standard protocol of MCPMARK (Wu et al., 2025b). For each task, we run the agent four times independently from the same starting snapshot and pass the resulting trajectories through the task's programmatic verifier. Pass@1 measures the average per-run success rate across the four runs, reflecting the agent's expected performance on a single attempt; Pass@4 measures whether *at least one* of the four runs succeeds, reflecting the agent's coverage when multiple attempts are permitted. Higher is better for both metrics.

A.2 Baseline Setup

In this section, we provide detailed descriptions of each baseline used in our comparison.

Tool-side Optimization. These baselines optimize tool descriptions or generate usage examples

MCP Service	Train Instances	#Tasks	Test Instances	#Tasks
Filesystem	file_context, file_property, student_database, threestudio	13	desktop, desktop_template, folder_structure, legal_document, papers, votenet	17
GitHub	claude-code, easyr1, harmony	14	build_your_own_x, mcpmark-cicd, missing-semester	9
Notion	it_trouble_shooting_hub, japan_travel_planner, standard_operating_procedure, team_projects	11	company_in_a_box, cs_student_dashboard, online_resume, python_roadmap, self_assessment, toronto_guide	17
Playwright	reddit	5	shopping, shopping_admin	14
PostgreSQL	dvdrental, lego, security	8	chinook, employees, sports, vectors	13
Total	*15 instances*	*51*	*21 instances*	*70*

Table 7: MCPMARK under the cross-environment split: entire environment instances are assigned to either train or test, so that no instance appears in both splits. The same 121 tasks as in Table 5 are repartitioned along the instance axis. Train instances are used for brainstorm-based memory construction; test instances evaluate transfer to unseen environment instances.

in an offline training phase, then inject the resulting artifacts into the agent prompt at inference time.

- **EasyTool** (Yuan et al., 2025) distills lengthy tool documentation into concise *tool instructions* through a two-stage pipeline: an LLM first rewrites each tool's documentation into a unified description of its general purpose and built-in functions, then generates scenario and parameter-value demonstrations verified by actually invoking the tool. We instantiate both stages with the same backbone LLM used in the corresponding experiment, and run the verification step on a clean environment snapshot to avoid contaminating evaluation states.
- **Play2Prompt** (Fang et al., 2025) optimizes tool documentation through a *tool-play* procedure that iteratively interacts with each tool and reflects on its outputs. It runs two beam-search stages: the first generates usage examples by exploring valid invocations in a trial-and-error loop, and the second refines the tool documentation using those examples as a validation set. We instantiate both stages with the same backbone LLM used in the corresponding experiment, and ground example generation in the current MCP environment instance to ensure valid invocations under its stateful resources.
- **ToolOptimal**[1] (Wu et al., 2025a) jointly optimizes the agent's task instruction and each tool's description through a three-stage verbalized loop: a feedback generator evaluates the rollout, a suggestion coordinator emits separate revisions for the instruction and the description, and a context refiner aggregates a batch of revisions before committing, analogous to gradient accumulation. We instantiate all three stages with the same backbone LLM used in the corresponding experiment, and feed the feedback generator both the verifier outcome (effectiveness) and the tool-call trajectory length and redundancy (efficiency) on the training split.

[1]The original paper does not name its method; we use ToolOptimal for brevity.

Experience-based Memory. These baselines accumulate experience from past task executions and inject it into the agent prompt for future tasks.

- **ExpeL** (Zhao et al., 2024) distills transferable *insights* from training trajectories through two channels: contrasting same-task success/failure pairs and aggregating patterns across multiple successful trajectories; each insight carries an importance count maintained by add/upvote/downvote/edit operations and is removed when the count reaches zero. Insight extraction is instantiated with the same backbone LLM used in the corresponding experiment. At inference, the full insight set is concatenated into the agent prompt, together with the top-k successful trajectories retrieved by task-embedding similarity.
- **A-MEM** (Xu et al., 2025) follows a Zettelkasten-inspired design in which each new memory triggers three operations: an LLM generates structured attributes and embeddings (note construction); retrieval identifies semantic neighbors and establishes bidirectional links (link generation); and linked notes update their representations to capture higher-order patterns (memory evolution). A-MEM was originally designed for long-conversation QA; to adapt it for MCP tool use, we replace dialogue content with stateful task-execution trajectories (task description, MCP tool-call sequence, and success/failure outcome), and use the new task description as the retrieval query.
- **ACE** (Zhang et al., 2025a) maintains an evolving *playbook* represented as a collection of struc-

tured, itemized bullets, each carrying content (a reusable strategy or failure mode) and counters tracking how often it was marked helpful or harmful. The playbook is updated through a three-role loop: a Generator runs the task and flags which bullets were useful or misleading, a Reflector distills lessons from the rollout, and a Curator merges those lessons into the playbook as incremental delta entries via deterministic non-LLM logic, with a separate grow-and-refine step that de-duplicates bullets by semantic similarity. At inference, the full playbook is provided to the Generator as context, rather than retrieving top-k entries. The Generator, Reflector, and Curator are all instantiated with the same backbone LLM used in the corresponding experiment. Since MCP verifiers provide deterministic pass/fail signals, we use them to drive the helpful/harmful counter bumps on the training split.

A.3 Agent Framework Setup

In this section, we detail the setups of our three adopted agent frameworks, ReAct, CodeAct, and SwitchAct:

A.3.1 ReAct

ReAct (Yao et al., 2023) interleaves natural-language reasoning with structured tool invocations, alternating between *thought* steps that plan the next action and *action* steps that issue a tool call. Its action space is restricted to single-tool invocations.

A.3.2 CodeAct

CodeAct (Wang et al., 2024b) unifies the action space into executable Python code: at each step the agent emits a Python snippet that may compose multiple tools, perform control flow, and post-process intermediate results, and observes the runtime output rather than the return value of a single function call.

A.3.3 SwitchAct

SwitchAct is a simple hybrid agent we implement to serve as the default backbone in our experiments: by combining both action interfaces above, it grants the agent comprehensive tool-use ability and allows memory to be accumulated from a broader spectrum of behaviors than either single-mode agent alone. Concretely, SwitchAct exposes *both* action interfaces to the LLM in parallel, letting the model autonomously choose, at each step, between a single-tool function call (as in ReAct) and a Python snippet (as in CodeAct). SwitchAct serves as the default agent backbone for our main results (§4.2); ReAct and CodeAct are used as target agents for the cross-agent generalization experiments in §4.3.

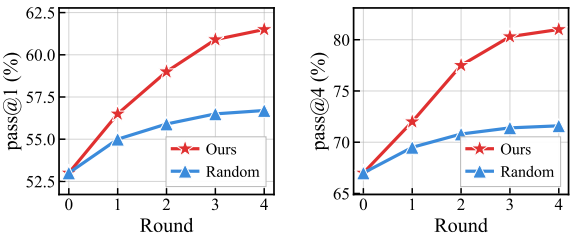

Figure 4: Effect of frontier exploration across brainstorm rounds on MCPMARK.

A.4 LLM Backbones

We evaluate with three proprietary LLM backbones spanning two model families: **GPT-5.4** (`gpt-5.4-20260305`), **GPT-5.4-mini** (`gpt-5.4-mini-20260317`), and **Grok-4** (`grok-4-2025-04-01-preview`). All three models are accessed through their respective official APIs. For the two GPT-5.4 backbones we set the reasoning effort to `medium` to control for inference-time compute; Grok-4 does not expose a reasoning-effort knob, and we use its default reasoning setting.

B Implementation Details

For Memory Bootstrapping, we generate k=3 seed tasks per tool and run N=4 rollouts per task. Capability Exploration runs for R=3 rounds; in each round, the proposer generates 3 boundary tasks and 3 affordance tasks for every target tool. At inference, Dynamic Memory Traversal retrieves k_r=3 seed trace nodes and operates under a read budget of B=8 steps.

C Additional Experiment Result

C.1 MCPMARK results using Grok-4

Table 8 presents the results on MCPMARK using the Grok-4 as the LLM backbone. `ToolAtlas` consistently outperforms the strongest baseline.

C.2 MCP-UNIVERSE results

Table 9 presents the results on MCP-UNIVERSE using GPT-5.4 as the LLM backbone. `ToolAtlas` consistently outperforms the strongest baseline.

Method	Filesystem		GitHub		Notion		Playwright		PostgreSQL		Overall	
	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑
Grok-4												
Vanilla	55.26	73.68	11.67	33.33	4.17	11.11	11.54	30.77	48.08	61.54	26.14	42.09
EasyTool	63.16	89.47	18.33	40.00	8.33	22.22	9.62	30.77	48.08	69.23	29.50	50.34
Play2Prompt	56.58	78.95	15.00	33.33	8.33	16.67	13.46	23.08	50.00	69.23	28.67	44.25
ToolOptimal	59.21	84.21	15.00	40.00	6.94	22.22	7.69	23.08	46.15	53.85	27.00	44.67
Expel	55.26	68.42	13.33	33.33	5.56	16.67	25.00	30.77	50.00	61.54	29.83	42.15
A-Mem	55.26	78.95	23.33	40.00	5.56	22.22	15.38	23.08	26.92	61.54	25.29	45.16
ACE	64.47	84.21	16.67	33.33	6.94	22.22	26.92	38.46	48.08	53.85	32.62	46.41
`ToolAtlas`	**73.68**	**94.74**	**26.67**	**46.67**	**15.28**	**27.78**	**28.85**	**46.15**	**53.85**	**76.92**	**39.67**	**58.45**
Impro(%)	*+14.29*	*+5.89*	*+14.32*	*+16.68*	*+83.43*	*+25.02*	*+7.17*	*+19.99*	*+7.70*	*+11.11*	*+21.61*	*+16.11*

Table 8: Performance comparison on MCPMARK under the same-environment split, using *SwitchAct* with Grok-4 as the backbone. The **best** and second best results are marked in bold and with underline, respectively. *Impro(%)* is the relative gain of `ToolAtlas` over the strongest baseline.

Method	Loc. Navigation		Financial Analysis		Web Searching		Overall	
	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑	pass@1↑	pass@4↑
Vanilla	30.77	46.15	77.50	83.33	41.22	59.46	49.83	62.98
EasyTool	35.58	53.85	73.33	83.33	47.30	59.46	52.07	65.55
Play2Prompt	32.69	50.00	71.67	86.67	45.95	62.16	50.10	66.28
ToolOptimal	31.73	50.00	75.00	86.67	46.62	62.16	51.12	66.28
Expel	39.42	46.15	78.33	86.67	42.57	64.86	53.44	65.89
A-Mem	37.50	53.85	71.67	83.33	45.95	59.46	51.71	65.55
ACE	42.31	57.69	79.17	90.00	43.24	62.16	54.91	69.95
`ToolAtlas`	**47.12**	**65.38**	**82.50**	**93.33**	**52.03**	**70.27**	**60.55**	**76.33**
Impro(%)	*+11.37*	*+13.33*	*+4.21*	*+3.70*	*+10.00*	*+8.34*	*+10.27*	*+9.12*

Table 9: Performance comparison on MCP-UNIVERSE under the same-environment split, using *SwitchAct* with GPT-5.4 as the backbone. The **best** and second best results are marked in bold and with underline, respectively. *Impro(%)* is the relative gain of `ToolAtlas` over the strongest baseline.

C.3 Frontier Exploration Effectiveness

Figure 4 compares the Frontier Exploration of `ToolAtlas` with a baseline exploration approach that uses the same per-round exploration budget but only samples from seed tasks.

Frontier exploration avoids redundant memory construction. Both methods improve as the number of brainstorm rounds R increases, but their trajectories differ sharply. Random improves early and then plateaus, while `ToolAtlas` continues to improve across rounds, widening the gap as R grows. This pattern reveals a failure mode we call *frontier blindness*: without an explicit estimate of what is already known about each tool, the LLM tends to propose overlapping tasks that revisit familiar tool behaviors. As a result, additional random exploration generates redundant experience. In contrast, `ToolAtlas` directs later rounds toward underexplored capabilities and boundary conditions, allowing each round to contribute new execution-verified knowledge. We set $R = 3$ by default, which captures most of the observed gain while keeping exploration cost moderate.

D Computational Resources and Software Environment

All experiments were executed on a CPU-only server featuring four Intel Xeon E7-4830 v3 processors (48 physical cores, 96 threads at 2.10 GHz base frequency) with 503 GB of main memory, running Ubuntu 24.04.1 LTS (kernel 6.8.0). Since `ToolAtlas` relies entirely on cloud-hosted LLM APIs for both brainstorm construction and task execution, no local GPU resources were required. The software environment was managed through Conda 25.9.1 with Python 3.12. Core API communication was handled by the OpenAI Python SDK (v1.109.1) and the MCP protocol library (v1.27.0). Benchmark evaluation on MCPMARK (v0.0.1) leveraged several environment-specific connectors, including Playwright (v1.58.0) for browser automation, `psycopg2` (v2.9.11) for PostgreSQL interaction, and `notion-client` (v2.4.0) for Notion API access. Token counting was performed with `tiktoken` (v0.12.0), and asynchronous HTTP requests were dispatched via `httpx` (v0.28.1). Data processing relied on NumPy (v2.4.4) and standard scientific Python utilities.

E The Use of Large Language Models

During the preparation of this manuscript, Large Language Models (LLMs) were employed to assist with language polishing, grammatical refinement, and coding support (e.g., debugging scripts and setting up experimental environments). Every piece of LLM-generated content was manually reviewed and validated by the authors prior to inclusion. The core research design, experimental analysis, and all scientific conclusions were carried out independently by the authors without LLM involvement.

F Case Study

Memory Bootstrapping

- **Target tool:** `list_directory_with_sizes`
- **Web search scenario:** ranking sibling folders by storage footprint
- **Generated task:** find the student folder(s) with the smallest and largest total stored bytes; write totals and names to `student_directory_size_extremes.txt`

Memory Induction for $\mathcal{G}_{\text{trace}}$

Task. Find the smallest and largest student folder by total bytes; write the result to `student_directory_size_extremes.txt`.

4 rollouts.

- Attempt 1 (×): 28 calls — looped between `list_directory` / `search_files` / `get_file_info`, took shallow sizes at face value.
- Attempt 2 (×): 36 calls — repeated `list_directory_with_sizes` four times on the same parent without summing recursively.
- Attempt 3 (×): 175+ calls — stuck in a `list_directory_with_sizes` loop, eventually wrote a wrong report.
- Attempt 4 (✓): 24 calls — `list_allowed_dirs` → `directory_tree` → `get_file_info` → `search_files` → `list_directory_with_sizes` → `get_file_info` → `write_file` → `read_text_file`.

1. Agent-neutral trace $\tilde{\xi}_i$ (per-task compression of the 4 rollouts)

1. `list_allowed_directories` — confirm the writable workspace before any inspection
2. `directory_tree` — survey top-level structure to enumerate candidate folders
3. `get_file_info` — validate target metadata before computing
4. `search_files` — recursive discovery so no candidate folder is missed
5. `list_directory_with_sizes` — obtain per-subdirectory totals for comparison
6. `get_file_info` — re-check exact byte totals on the identified extremes
7. `write_file` — write the summary file
8. `read_text_file` — verify the saved note matches the intended output

Memory Induction for $\mathcal{G}_{\text{cap}}$

Input. All query nodes whose refined trace touches `list_directory_with_sizes`. For each such query we feed in the refined trace plus any `task_tips` mentioning the tool:

Output (new fields appended to `tools["list_directory_with_sizes"]`):

```
Input:

[q_student_directory_size_extremes_note] summary: rank sibling folders by total bytes...
  Refined trace:
    1. list_allowed_directories - confirm writable workspace
    2. directory_tree          - enumerate candidate folders
    3. get_file_info          - validate target metadata
    4. search_files           - recursive discovery
    5. list_directory_with_sizes - obtain per-subdirectory totals
    6. get_file_info          - re-check exact byte totals on extremes
    7. write_file             - write summary
    8. read_text_file         - verify saved output
  Related tips: shallow per-entry sizes are not recursive totals;
           verify extremes with get_file_info before writing.
[q_. . .  ] . . .

Output:

{
  "new_affordance": [
```

```
    {"text": "Sum child entries explicitly when ranking siblings — the reported sizes are
shallow.",
     "source_queries": ["q_student_directory_size_extremes_note", . . . ]}
  ],
  "new_boundaries": [
    {"text": "Per-entry sizes are non-recursive; do not treat them as folder totals.",
     "source_queries": ["q_student_directory_size_extremes_note"]}
  ],
  "new_co_usage": [
    {"text": "list_directory_with_sizes → get_file_info to confirm exact byte totals on
extremes.",
     "related_tools": ["list_directory_with_sizes", "get_file_info"],
     "source_queries": ["q_student_directory_size_extremes_note"]}
  ]
}
```

Memory Induction for $\mathcal{G}_{\text{strat}}$

Input. A subgraph of semantically related query nodes (here: workspace-rooted file-system reporting tasks). For each query we feed in the refined trace plus its `task_tips`:

Output (new entry appended to `strategy`):

```
Input:

[q_student_directory_size_extremes_note] summary: rank sibling folders by total bytes
  Refined trace: list_allowed_directories → directory_tree → get_file_info →
                 search_files → list_directory_with_sizes → get_file_info →
                 write_file → read_text_file
  Task tips: start by confirming the writable workspace; only then explore.
[q_largest_log_file_per_service]   Refined trace: list_allowed_directories → directory_tree →
. . .  ; tips: . . .
[q_count_python_files_under_assets] Refined trace: list_allowed_directories → search_files →
. . .  ; tips: . . .
[q_. . .  ] . . .

Output:

{
  "new_strategy": [
    {"text": "Begin with workspace and constraint discovery before acting: identify allowed roots
and writable locations first so later steps target the correct scope.",
     "source_queries": ["q_student_directory_size_extremes_note",
                        "q_largest_log_file_per_service",
                        "q_count_python_files_under_assets", . . . ]}
  ]
}
```

Capability Exploration

```
Sampled knowledge-graph neighborhood (centered on list_directory_with_sizes):

nodes:
  n1 = Tool{name: list_directory_with_sizes, role: center}
  n2 = Tool{name: get_file_info, role: verifier}
  n3 = Tool{name: directory_tree, role: structure_survey}
  n4 = Tool{name: search_files, role: recursive_discovery}

typed edges:
  (n1) -[co_usage: confirm exact byte totals]-→ (n2)
  (n3) -[co_usage: enumerate candidate folders]-→ (n1)
  (n4) -[co_usage: discover recursive candidates]-→ (n3)
  (n4) -[co_usage: verify discovered paths]-→ (n2)

graph triples shown to proposer:
  ⟨list_directory_with_sizes, co_usage, get_file_info⟩
  ⟨directory_tree, co_usage, list_directory_with_sizes⟩
  ⟨search_files, co_usage, directory_tree⟩
  ⟨search_files, co_usage, get_file_info⟩

Extracted knowledge from the subgraph nodes (the knowledge_map actually shown to the proposer LLM):

## list_directory_with_sizes
```

```
  affordance:
    - Sum child entries explicitly when ranking siblings — sizes are shallow.
  boundaries:
    - Per-entry sizes are non-recursive; do not treat them as folder totals.
  co_usage:
    - list_directory_with_sizes → get_file_info to confirm exact byte totals on extremes.

## get_file_info
  affordance:
    - Authoritative byte counts on a single path.
    - . . .
  boundaries: . . .
  co_usage:   . . .

## directory_tree
  affordance: . . .
  boundaries: . . .
  co_usage:
    - Pair with list_directory_with_sizes / get_file_info for recursive discovery.
    - . . .

## search_files
  affordance: . . .
  boundaries: . . .
  co_usage:   . . .
```

Two proposed tasks (same target tool, opposite directions):

- **R2 — boundary**: On a workspace with 10k+ deeply-nested subfolders and symlink cycles, rank the top-5 folders by *recursive* byte total using `list_directory_with_sizes`; write the ranking to `deep_tree_rank.txt`. *(Goal: push the shallow-size limitation until it breaks.)*
- **R3 — affordance**: Using `list_directory_with_sizes` together with `get_file_info`, identify the single largest *file* (not folder) under each top-level student directory and write `largest_file_per_student.txt`. *(Goal: exercise an untested file-level use combined with the known co-usage partner.)*

Dynamic Memory Traversal

New task (held-out). *For every top-level student folder, find the file with the most recent modification time and write* `path<TAB>mtime` *lines to* `latest_file_per_student.txt`.

Step 1 — embedding match to seeds
Encode the task summary, retrieve top-3 query nodes by cosine similarity:

Step 2 — navigator tool calls

1. `ReadTrace` with argument `q_student_directory_size_extremes_note` — refined trace + task_tips
2. `Expand` with argument `q_student_directory_size_extremes_note` — similar query `q_largest_log_file_per_service`
3. `ReadTrace` with argument `q_largest_log_file_per_service` — trace exhibiting per-group extremum pattern
4. `ReadTool` with argument `get_file_info` — affordance entries (authoritative byte/mtime on a single path)
5. `ReadStrategy` with argument `q_student_directory_size_extremes_note` — strategy i_{001} (workspace discovery first)

Step 3 — emitted guidance (returned to executor)

```
SEED CANDIDATES (top-3 by embedding similarity):
  - q_student_directory_size_extremes_note   (sim=0.81): rank sibling folders by total bytes
  - q_largest_log_file_per_service      (sim=0.73): per-group extremum over a file attribute
  - q_count_python_files_under_assets     (sim=0.61): recursive per-folder aggregation

Past-task playbook:
  1. list_allowed_directories — confirm writable workspace
```

```
  2. directory_tree          – enumerate top-level student folders
  3. search_files           – recursive file discovery under each folder
  4. get_file_info         – read mtime per file (authoritative attribute)
  5. write_file            – write path<TAB>mtime lines
  6. read_text_file        – verify saved output

Strategy: begin with workspace and constraint discovery before acting (i_001).
Tool tips:
  - get_file_info → authoritative byte/mtime on a single path; use to resolve per-group extrema.
  - list_directory_with_sizes → not needed here (per-entry sizes non-recursive, and task is
file-level).
```

Step 4 — execution

Executor follows the guidance and finishes in a single pass; verifier accepts byte-exact.

1. `list_allowed_directories` — Strategy i_{001} — workspace discovery first
2. `directory_tree(/workspace/students)` — Playbook step 2 — enumerate top-level student folders
3. `search_files(student_01)` — Playbook step 3 — recursive file discovery
4. `get_file_info(...)` $\times N$ — Tool tip — `get_file_info` is the authoritative source for mtime
5. (repeat 3–4 for each student folder) — per-group extremum pattern carried over from `q_largest_log_file_per_service`
6. `write_file(latest_file_per_student.txt)` — Playbook step 5
7. `read_text_file(latest_file_per_student.txt)` — Playbook step 6 — verify saved output

Effect of guidance. `list_directory_with_sizes` — the tool that broke 3/4 rollouts in the seed round — never appears, because the Tool-tips line explicitly excluded it for this file-level variant. The executor also skips the shallow size-summing loop that consumed 175+ calls in seed-round attempt 3.

G Prompt Set

This section lists the core prompts used by `ToolAtlas`, grouped by the methodological stage they support: seed task proposal in Memory Bootstrapping, probe task proposal in Capability Exploration, trace reflection and capability/strategy aggregation in Memory Induction, and graph navigation in Dynamic Memory Traversal.

Memory Bootstrapping Prompt

You are a task designer for a benchmark that evaluates LLM agents on MCP tool usage.
Goal: Propose ONE new seed task that is feasible, non-trivial, covers a representative use of the target tool `{target_tool.name}`, and is best solved by it.
Input:
- Target tool: `{target_tool.name}`, with description {target_tool.description} and input schema {target_tool.schema}
- Real-world usage snippets retrieved via web search for grounding: {web_search_context}
- One few-shot example for format reference (do NOT copy): {few_shot_task}

Output (JSON):
- `task_id`: snake_case identifier
- `summary`: one-sentence summary
- `tool_used`: `{target_tool.name}`
- `description_md`: Markdown task description with Title, Task Description, Objectives, Constraints, and Expected Output
- `verify_py`: self-contained Python verifier that exits 0 on success or 1 on failure

Return ONLY the JSON object.

Tool-Trace Graph Induction Prompt

You are analyzing execution traces from a tool-using agent to extract reusable knowledge.
Goal: Convert raw rollouts of a task into an `agent_neutral_trace` and `task_level_tips`, which together form a trace node $q_{ij} = (Q_{ij}, \tilde{\xi}_{ij}, \mathcal{P}_{ij})$.
Input:
- Task description: {task_summary}
- Category: {category}
- Rollouts from multiple attempts: {rollouts_text}
- Real MCP tool namespace and allowed tool names: {tool_namespace_hint}

Abstraction rule: Produce environment-invariant and agent-neutral knowledge. Do NOT mention specific filenames, page IDs, table names, issue numbers, schema fields, literal values, paths, or attempt indices, and do NOT preserve agent-specific action syntax, retry patterns, or reasoning style. **Output (JSON):**
- `agent_neutral_trace`: an ordered sequence of `(tool, rationale)` pairs synthesized across all rollouts, using the most successful attempt as the backbone when available and incorporating useful corrections from failed or partial attempts. `tool` is a real MCP tool name; `rationale` captures the tool-use intent of the step.
- `task_level_tips`: reusable task-level learnings distilled from successful and failed rollouts, not tied to one step. Rewrite failure lessons as positive cautions.

```
{
"agent_neutral_trace": [{"tool": "<real_mcp_tool>",
"rationale": "<tool-use intent of this step>"}],
"task_level_tips": ["<env-invariant tip>"]
}
```
Return ONLY the JSON object.

Tool-Capability Graph Induction Prompt

You are analyzing how tool `{tool_name}` has been used across multiple tasks.
Goal: Update the Tool-Capability Graph $\mathcal{G}_{\text{cap}}$ by inducing new affordance entries $\mathcal{U}_i$, boundary entries $\mathcal{B}_i$, and co-usage entries $\mathcal{C}_i$ for tool node t_i from usage evidence.
Input:
- Tool description: {tool_description}
- Usage evidence from {n_queries} tasks: {evidence_text}
- Existing affordance entries to avoid duplicates: {existing_affordance}
- Existing boundary entries to avoid duplicates: {existing_boundaries}
- Existing co-usage entries to avoid duplicates: {existing_co_usage}

Abstraction rule: Produce environment-invariant entries grounded in usage evidence. Do NOT reference specific filenames, page IDs, table names, issue numbers, schema

fields, literal values, paths, or task identifiers inside the entry text. **Output (JSON):**

- `new_affordance`: valid uses that t_i reliably supports.
- `new_boundaries`: regimes where t_i becomes unreliable or infeasible.
- `new_co_usage`: recurrent compositions with peer tools.

```
{
"new_affordance":          [{"text":
"<affordance>",
"source_queries": ["q1","q2"]}],
"new_boundaries":          [{"text":
"<boundary>",
"source_queries": ["q1"]}],
"new_co_usage": [{"text": "<co-usage
pattern>",
"related_tools":
["tool_a","tool_b"],
"source_queries": ["q1","q2"]}]
}
```

Only include entries backed by usage evidence. Every entry must cite `source_queries`. If there is nothing new to add, return empty lists.

Tool-Strategy Graph Induction Prompt

You are analyzing task-level tips from multiple tasks to find recurring planning patterns. **Goal:** Update the Tool-Strategy Graph $\mathcal{G}_{\text{strat}}$ by distilling recurring task-level tips into strategy nodes that capture high-level orchestration principles. **Input:**

- Task-level tips collected from {n_queries} tasks: {all_tips_text}
- Existing strategy nodes to avoid duplicates: {existing_strategies}
- Minimum recurrence threshold: {threshold} source queries

Selection rules:

- Keep only planning patterns that recur across multiple tasks; discard single-tool properties.
- Prefer high-level orchestration principles over execution-level tool tips.
- Each strategy node must be backed by at least {threshold} source queries.
- Do not repeat existing strategy nodes.

Output (JSON):

- `new_strategy`: recurring high-level orchestration principles with supporting query IDs.

```
{
"new_strategy":            [{"text":
"<orchestration principle>",
"source_queries": ["q1","q2","q3"]}]
}
```

Only include genuinely new strategy nodes. Return an empty list if nothing qualifies.

Dynamic Memory Traversal Prompt

You are the navigator in Dynamic Memory Traversal. You have access to a tool memory graph $\mathcal{M}_{\mathcal{T}} = (\mathcal{G}_{\text{trace}}, \mathcal{G}_{\text{cap}}, \mathcal{G}_{\text{strat}})$ induced from past task rollouts. **Goal:** Given a NEW TASK Q, perform a graph walk over $\mathcal{M}_{\mathcal{T}}$ and compress the gathered context into task-conditioned guidance $g(Q)$ for the user agent.

Traversal operations:

- `ReadTrace(q)`: read the task, agent-neutral trace, and task-level tips of trace node q.
- `Expand(q)`: follow trace-graph edges to similar trace nodes.
- `ReadTool(t)`: read affordance, boundary, and co-usage entries from tool node t.
- `ReadStrategy(q)`: read strategy nodes supported by trace node q.
- `Done`: stop traversal and emit the final task-conditioned guidance $g(Q)$.

Traversal policy:

- Start from k_r nearest Tool-Trace nodes as the initial context C_0.
- Use graph neighbors before global search.
- Operate under a fixed read budget; call `Done` when the context is sufficient or the read budget is exhausted.
- Each operation call MUST include a `thought` field explaining the choice.

Output (JSON via `Done`): the emitted guidance $g(Q)$ has four sections, each grounded in the corresponding layer of $\mathcal{M}_{\mathcal{T}}$.

- `seed_candidates`: the initial Tool-Trace seeds, each with its qid, embedding similarity to Q, and a one-line summary.
- `playbook`: an ordered sequence of `(tool, rationale)` pairs synthesized from retrieved agent-neutral traces and adapted to Q. Each `tool` must be a real MCP tool name.
- `strategy`: applicable orchestration principles drawn from strategy nodes in $\mathcal{G}_{\text{strat}}$, each with its source strategy-node id.
- `tool_tips`: per-tool hints relevant to Q, drawn from affordance, boundary, and co-

usage entries; include negative tips (*when not to use*) when boundary entries apply.

```
{
"seed_candidates": [{"qid": "<qid>",
"sim": <float>,
"summary": "<one-line summary>"}],
"playbook":                [{"tool":
"<real_mcp_tool>",
"rationale": "<role of this step for
Q>"}],
"strategy":                [{"text":
"<orchestration principle>",
"source_strategy_id":
"<strategy_node_id>"}],
"tool_tips":               [{"tool":
"<real_mcp_tool>",
"tip": "<task-relevant hint>"}]
}
```

Omit generic advice. Never recommend unavailable tools. If no past task is relevant, return empty lists for all four sections.