PREPING: Zero-Task Agent Memory Construction via Self-Practice

Yumin Choi[1] **Sangwoo Park**[1] **Minki Kang**[1] **Jinheon Baek**[1] † **Sung Ju Hwang**[1,2] †

[1]KAIST [2]DeepAuto.ai

{yuminchoi, jinheon.baek, sungju.hwang}@kaist.ac.kr

Abstract

Agent memory is typically constructed either offline from curated demonstrations or online from post-deployment interactions. However, regardless of how it is built, an agent faces a cold-start gap when first introduced to a new environment without any task-specific experience available. In this paper, we study *pre-task memory construction*: whether an agent can build procedural memory before observing any target-environment tasks, using only self-generated synthetic practice. Yet, synthetic interaction alone is insufficient, as without controlling what to practice and what to store, synthetic tasks become redundant, infeasible, and ultimately uninformative, and memory further degrades quickly due to unfiltered trajectories. To overcome this, we present PREPING, a proposer-guided memory construction framework. At its core is *proposer memory*, a structured control state that shapes future practice. A *Proposer* generates synthetic tasks conditioned on this state, a *Solver* executes them, and a *Validator* determines which trajectories are eligible for memory insertion while also providing feedback to guide future proposals. Experiments on AppWorld, BFCL v3, and MCP-Universe show that PREPING substantially improves over a no-memory baseline and achieves performance competitive with strong playbook-based methods built from offline or online experience, with deployment cost 2.99× lower on AppWorld and 2.23× lower on BFCL v3 than online memory construction. Further analyses reveal that the main benefit does not come from synthetic volume alone, but from proposer-side control over feasibility, redundancy, and coverage, combined with selective memory updates.

1 Introduction

LLM agents are increasingly deployed to solve tasks by acting in executable environments, from tool APIs and Model Context Protocol (MCP) servers to command-line interfaces [2, 12, 15, 31, 36, 41]. In these environments, success requires more than knowing which tools are available: agents must learn environment-specific procedures, including how tool calls compose, which preconditions matter, and how to recover from state-dependent failures [23, 25, 26]. Reusable memory offers a practical pathway to capture such procedural knowledge as tool-use guidance, workflow rules, and playbook-style instructions. Prior work on agent memory shows that such memory can substantially improve downstream execution performance when useful task experience is available [5, 17, 28, 39].

However, existing memory construction methods typically rely on target-environment task experience, either collected before deployment as demonstrations or trajectories, or accumulated after deployment from user interactions [20, 32, 46]. This requirement creates a practical gap for newly connected environments. Offline construction depends on humans to design, collect, or solve tasks per environment, which is costly and rarely available before deployment. Online construction avoids this upfront

†Equal advising. Project page.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

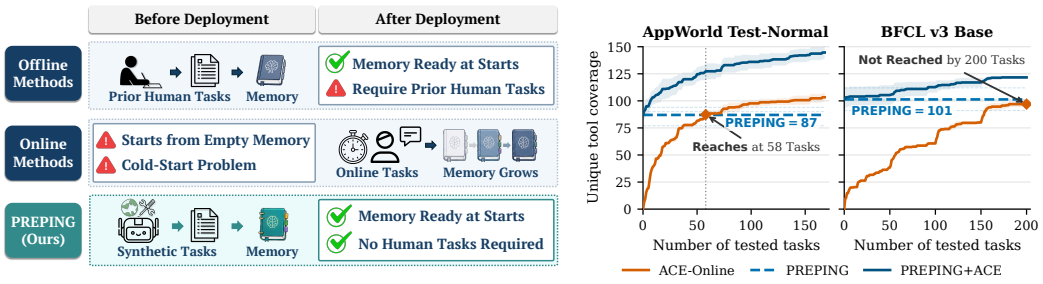

Figure 1: **PREPING builds memory before the first user task and mitigates online cold-start.** Left: Unlike offline methods that require prior human tasks or online methods that start from empty memory, PREPING constructs procedural memory through self-generated synthetic practice before deployment. Right: PREPING establishes broad tool coverage before deployment, whereas ACE-Online must accumulate coverage from user-facing tasks, requiring 58 tasks to match PREPING on AppWorld and still falling short after 200 tasks on BFCL v3.

effort but starts from empty memory: the agent learns only after user-facing tasks arrive, exposing users to early failures, memory-update latency, and additional deployment-time costs. Memory is thus most needed precisely when the experience required to build it has not yet been collected.

Motivated by this, we study *pre-task memory construction*: building reusable procedural memory before any target-environment task data is available. In this setting, the agent may inspect environment documentation, execute tools, and observe their outputs, but it has not seen human-provided tasks, demonstrations, solved trajectories, or deployment-time user interactions from the target environment. This makes the setting distinct from simply performing online memory construction earlier: without task instructions, the agent lacks a direct signal about which user goals will appear, which tools should be composed, or what successful task-level workflows should look like.

Pre-task memory construction is challenging because access to the environment alone does not reveal reusable task-level procedures. Tool documentation and schemas specify callable interfaces, but often leave environment-specific preconditions, state-dependent constraints, and failure recovery strategies implicit; likewise, free-form exploration can reveal isolated tool-execution examples, but does not reliably produce reusable procedures for accomplishing task-level goals. The agent must therefore create and execute its own task-level objectives through self-generated synthetic practice. However, naive synthetic practice introduces two coupled control problems: tasks can be redundant, infeasible, or poorly grounded in the environment, and storing their trajectories can contaminate memory with misleading guidance. Therefore, pre-task memory construction is not merely a synthetic task generation problem, but a problem of jointly shaping *what to practice* and *what to store*.

To address this, we introduce PREPING (**Pre**-Task **RE**usable **P**laybook Mak**ING**), a framework that couples proposer-guided synthetic practice with validation-gated memory admission. PREPING maintains *proposer memory*, a construction-time state, which tracks prior synthetic practice history and environment information. Conditioned on this state, a *Proposer* generates feasible synthetic tasks that expand coverage toward under-explored aspects of the environment, and a *Solver* executes these tasks to produce trajectories. A *Validator* then filters out infeasible task trajectories before memory insertion, and a memory update module distills the remaining trajectories into *solver memory*, the reusable procedural memory used for future tasks. In this way, PREPING builds memory by shaping both the practice distribution and the quality of trajectories admitted into memory.

We evaluate PREPING on AppWorld [30], BFCL v3 [24], and MCP-Universe [12], covering stateful app execution, structured function calling, and realistic MCP-server tool use. The results show three main findings. First, PREPING builds effective pre-task memory across all benchmarks, improving over a no-memory baseline by 17.1 points on AppWorld, 19.3 points on BFCL v3, and 5.4 points on MCP-Universe, while remaining competitive with methods that rely on human-defined or deployment-time target tasks, even though PREPING requires no such target-task experience. Second, ablations show that these gains come not from synthetic task generation alone, but from validation-gated memory admission and proposer-side control over feasibility, tool coverage, and downstream relevance. Third, PREPING offers deployment benefits as an initialization for the online memory construction approach (ACE [41]) and as frozen pre-task memory. In particular, PRE-

PING+ACE improves performance on AppWorld from 71.3 to 76.3 while reducing early cold-start failures and tool-coverage shortfall (Fig. 1, Right); when frozen, PREPING avoids deployment-time memory-update calls, reducing cost by $2.99\times$ on AppWorld and $2.23\times$ on BFCL v3 relative to ACE-Online.

2 Related Work

Memory for LLM Agents. Reusable memory enables LLM agents to adapt across tasks while keeping the underlying model fixed [7, 35, 40]. As external context, memory can be inspected, revised, and transferred across models or modules, making it attractive for tool-using agents and compound AI systems [9, 14, 41]. Prior work stores experience in various forms, including persistent memory, workflow knowledge, playbook-style guidance, or long-term context distilled from prior interactions [11, 21, 22, 45]. Agent Workflow Memory induces reusable workflows from successful trajectories and retrieves them for future task solving [32], while ACE grows playbook-style context through structured generation, reflection, and curation from offline or online task feedback [41]. These methods show that external memory can improve downstream execution by capturing environment-specific procedures, failure modes, and task-solving strategies [13, 27, 42, 44]. However, a key commonality is that memory is constructed from target-environment task experience, whether as curated demonstrations, logged trajectories, successful workflows, or online user interactions [17, 20, 46]. This assumption is limiting for newly connected environments, where such experience may not yet exist. In contrast, our work studies the preceding cold-start phase: constructing reusable procedural memory before any human-provided or deployment-time target tasks are available.

Self-Generated Practice for Policy Updates. A separate line of work uses self-generated tasks, self-play, and automatic curricula to improve agent policies or model behavior without human annotations [4, 43, 48]. In the tool-use setting, Zhou et al. [47] instantiate this pattern with a challenger that interacts with tools to generate Code-as-Task problems with executable verification functions, and an executor that is optimized with evaluation feedback as reward. Huang et al. [8] develop a related co-evolution loop for reasoning, where a Challenger is rewarded for producing tasks near a Solver's capability frontier and the Solver is trained on filtered self-generated problems. Other self-evolving systems follow related patterns in search, tool-integrated reasoning, software engineering, and corpus-grounded reasoning [1, 10, 33, 34, 37, 38]. These methods demonstrate the value of self-generated practice, but mainly as a training signal for policy or model updates, with task generation optimized for difficulty, solvability, curriculum progression, executable verification, or reward quality [6, 16]. In contrast, our setting requires a different form of control: since the goal is to construct reusable textual memory, synthetic practice must expose broad, non-redundant, and environment-grounded procedures, while only trajectories suitable for distillation should be admitted into memory. Therefore, our setting is not only about generating challenging or verifiable tasks, but about jointly controlling what to practice and what to store so that synthetic experience becomes deployable procedural guidance.

3 Method

We propose PREPING, a framework for pre-task memory construction that turns environment access (before any target task experience) into procedural memory through controlled synthetic practice.

3.1 Pre-Task Memory Construction

We first formalize the pre-task memory construction setting. Given a target environment E (executable) and its documentation $\mathcal{D}$, a construction procedure may inspect $\mathcal{D}$, call tools in E, and observe the resulting environment feedback, but it has no access to target-environment task experience, such as human-provided task instructions, demonstrations, solved trajectories, or logged user interactions. The output is a solver memory M_{sol} that is supplied to the agent at deployment time.

This setting differs from standard offline or online memory construction because the construction procedure has no access to the task distribution. Documentation specifies callable interfaces, but rarely reveals which tool compositions, preconditions, or failure modes will matter for downstream. The agent must therefore actively produce its own task-level objectives, execute them in the environment,

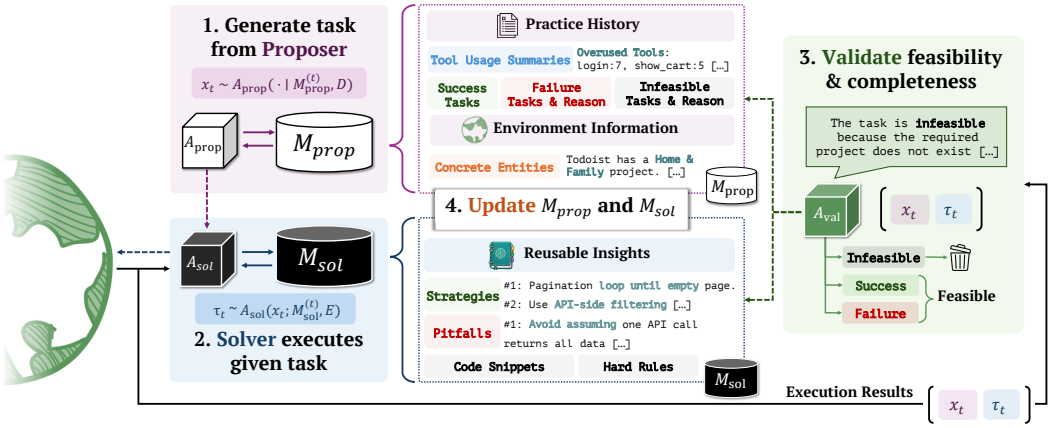

Figure 2: **Overview of PREPING.** PREPING builds procedural memory before deployment through a synthetic-practice loop: proposing tasks, executing them in the environment, validating the resulting trajectories, and updating memory. Proposer memory (M_{prop}) shapes what to practice next, while the Validator filters which synthetic trajectories are reliable enough to become solver memory (M_{sol}).

and convert the resulting experience into memory. PREPING treats this as a controlled synthetic-practice problem, where the challenge is to jointly regulate what to practice and what to store.

3.2 PREPING: Controlled Synthetic Practice for Pre-Task Memory Construction

PREPING separates the construction process into two memory states with distinct roles. Proposer memory (M_{prop}) is a construction-time control state that guides future task proposals. It records what has already been practiced, which tools or workflows remain under-explored, and which proposals failed due to infeasibility or poor grounding. In contrast, solver memory (M_{sol}) is the deployment-facing procedural memory that will later be provided to the task-solving agent. This separation is important because signals useful for controlling practice, such as rejected tasks, repeated tool use, and coverage imbalance, should not necessarily be exposed as procedural guidance during deployment.

At each construction iteration, PREPING coordinates three LLM-powered modules instantiated with different roles and contexts: a Proposer (A_{prop}), a Solver (A_{sol}), and a Validator (A_{val}). At iteration t, the Proposer generates a synthetic task x_t conditioned on documentation and proposer memory; the Solver executes x_t in the environment to produce a trajectory τ_t; and the Validator evaluates whether the task-trajectory pair is feasible, grounded, and useful for memory construction, as follows:

$$x_t \sim A_{\text{prop}}(\cdot \mid M^{(t)}_{\text{prop}}, \mathcal{D}), \qquad \tau_t \sim A_{\text{sol}}(\cdot \mid x_t, M^{(t)}_{\text{sol}}, E), \qquad v_t = A_{\text{val}}(x_t, \tau_t).$$

The two memories are then updated asymmetrically, as follows:

$$M^{(t+1)}_{\text{prop}} = U_{\text{prop}}(M^{(t)}_{\text{prop}}, x_t, \tau_t, v_t), \qquad M^{(t+1)}_{\text{sol}} = \begin{cases} U_{\text{sol}}(M^{(t)}_{\text{sol}}, x_t, \tau_t, v_t), & \text{if Feasible}(v_t), \\ M^{(t)}_{\text{sol}}, & \text{otherwise.} \end{cases}$$

U_{prop} and U_{sol} denote the proposer-memory and solver-memory update rules, and $\text{Feasible}(v_t)$ indicates that the synthetic task and its trajectory are grounded in the environment and suitable for memory construction. This asymmetric update is the core design of PREPING: all experience (including rejected tasks) updates proposer memory and shapes future practice, while only feasible task-trajectory pairs are eligible for solver memory. For clarity, while equations show one synthetic task per iteration, in practice, PREPING samples a batch of tasks as shown in Alg. 1.

3.3 Proposer Memory Controls What to Practice

The first control decision is *what to practice next*. Synthetic tasks determine which parts of the environment will be exercised and, ultimately, which procedures can be distilled into memory. If proposals repeatedly target the same tools, APIs, entities, or workflows, construction yields redundant

memory with limited downstream coverage. If proposals depend on unsupported entities, unavailable tools, or hidden preconditions, they produce infeasible trajectories that provide little reusable signal. We therefore use proposer memory (M_{prop}) as a construction-time control state to make task proposal history-aware, coverage-seeking, and grounded in the executable environment.

Proposer memory contains two complementary views of prior practice. The first is a *practice-history view*, which records previous synthetic tasks, tools or APIs invoked in their trajectories, validation outcomes, and failure or infeasibility reasons. It also maintains aggregate usage summaries, identifying which tools, functions, or workflows have been over-practiced or under-practiced. Operationally, U_{prop} extracts invoked tools and functions from trajectories using rule-based parsers and combines them with validator feedback. The second is a *grounded-environment view*, which records concrete entities, observed states, preconditions, and constraints discovered during execution. These observations are summarized with an LLM so that future proposals can refer to executable environment facts rather than inventing unsupported task details. When rendered as context for A_{prop}, these two views impose complementary pressures: practice history discourages near-duplicate tasks and encourages expansion toward under-covered parts of the environment, while grounded environment information keeps that expansion feasible. As construction proceeds, M_{prop} therefore acts as a control state over the synthetic practice distribution, rather than a passive log of previous attempts.

3.4 Validator-Gated Memory Controls What to Store

The second control decision is *what to store*. Since synthetic practice is produced without human-written task specifications or gold trajectories, its outputs are not reliable sources of memory by default. The proposed tasks may be infeasible, depend on missing environment state, require unavailable tools, or only partially specify the intended objective. If their trajectories are inserted into solver memory without filtering, synthetic artifacts can be distilled into misleading procedural guidance.

To prevent this, Validator (A_{val}) evaluates each task-trajectory pair and produces a signal v_t with feasibility and task-completion scores, along with rationales. The feasibility judgment checks whether the proposed task is grounded in the environment and executable under the observed state and available tools, while the completion judgment checks whether the Solver accomplishes the intended synthetic objective. These two judgments serve different roles: feasibility gates solver-memory insertion, while completion guides what procedural lesson, if any, should be distilled from the trajectory.

These Validator outputs are used in three ways. First, they gate solver-memory updates: infeasible pairs are excluded from U_{sol}. Second, all validation outcomes, including rejected pairs, are passed to U_{prop}, helping future proposals avoid repeated failure modes. Third, for admitted pairs, U_{sol} converts the task, trajectory, and validation outcome into compact procedural bullets (rather than appending raw interaction logs), following a reflector-curator style playbook induction process in ACE [41].

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate PREPING on three complementary agent benchmarks: AppWorld [30], BFCL v3 [24], and MCP-Universe [12], which span diverse forms of executable agent environments: stateful application workflows, structured function calling, and realistic MCP-server interactions. AppWorld tests stateful application tasks, where agents write code against app APIs (e.g., Spotify) and are scored by a state-based evaluator that checks the final environment state. We report AppWorld on Test-Normal (N), a held-out split drawn from the same distribution as the offline training split, and Test-Challenge (C), a harder split whose tasks require at least one unseen app. For metrics, Task Goal Completion (TGC) is the percentage of tasks for which all evaluation tests pass, while Scenario Goal Completion (SGC) credits a scenario only when all of its task variants are solved. In the main table, N-TGC/N-SGC and C-TGC/C-SGC denote TGC/SGC on Test-Normal and Test-Challenge, respectively. BFCL v3 tests executable function calling under schema and dialogue constraints; we report the Base, Long Context (Ctx.), Missing Parameter (Para.), and Missing Function (Func.) categories. MCP-Universe tests tool use over real Model Context Protocol servers with heterogeneous tool inventories and execution-based evaluators. We use four MCP-Universe categories: Repository Management (Repo.), Financial Analysis (Fin.), 3D Designing (3D.), and Browser (Brow.).

Table 1: **Main results.** Pre-Task Methods construct memory before any task data is available, while Task-Informed Methods use task data available before or during deployment. Scores are averaged over three independent runs. Bold numbers indicate the best pre-task method.

	AppWorld					BFCL v3					MCP-Universe				
Method	**N-TGC**	**N-SGC**	**C-TGC**	**C-SGC**	**Avg.**	**Base**	**Ctx.**	**Para.**	**Func.**	**Avg.**	**Repo.**	**Fin.**	**3D.**	**Brow.**	**Avg.**
Pre-Task Methods															
Base	69.6	49.4	56.7	36.7	53.1	43.7	40.0	23.0	28.3	33.8	8.1	59.2	29.8	31.3	32.1
Direct Memory	72.8	55.4	56.0	34.5	54.7	43.2	42.3	22.3	29.3	34.3	7.1	67.5	26.3	30.2	32.8
Random Exploration	73.6	52.4	55.9	36.9	54.7	59.0	51.3	32.8	41.3	46.1	10.1	60.8	22.8	**38.5**	33.1
Guided Exploration	74.6	56.0	56.6	35.0	55.6	55.5	51.2	28.5	40.3	43.9	**12.1**	61.7	33.3	31.3	34.6
PREPING	**83.7**	**70.2**	**72.2**	**54.7**	**70.2**	**65.2**	**59.3**	**37.8**	**50.2**	**53.1**	10.1	**70.8**	**36.8**	32.3	**37.5**
Task-Informed Methods															
ACE-Offline	82.7	69.1	69.0	50.3	67.8	–	–	–	–	–	–	–	–	–	–
ACE-Online	80.4	65.5	78.3	60.9	71.3	62.3	55.0	36.7	52.3	51.6	14.1	69.2	43.9	37.5	41.2

Pre-Task Methods. We compare methods that operate strictly within the pre-task setting, where no target-environment task data are available. **Base** uses no constructed memory and solves downstream tasks directly from the model and environment context. **Direct Memory** constructs memory directly from environment documentation without execution, by sampling and combining diverse subsets of API or tool documentation into memory. We also evaluate execution-based baselines that construct memory from free-form environment interaction without task-level objectives. Specifically, **Random Exploration** prompts the agent to explore the environment without additional constraints, while **Guided Exploration** conditions exploration on prior exploration history to encourage under-explored APIs or tools. **PREPING** instead constructs memory from proposer-guided synthetic task practice: it generates task-level objectives, executes them in the environment, and admits only validator-approved task-trajectory pairs into solver memory. All memory-construction methods use the same reflector-curator memory induction pipeline [41], isolating whether memory is induced from documentation, free-form exploration, or validated synthetic-task practice. We provide baseline details in Sec. A.5.

Task-Informed Methods. We also report task-informed memory construction methods as reference points. Unlike the pre-task methods above, these methods are allowed to use target-environment task data, and therefore assume information that is unavailable in the pre-task setting. **ACE-Offline** [41] constructs memory before deployment from human-defined target tasks and their execution feedback. This setting can produce strong memory when a representative task set is available, but it requires task collection or task design for each new environment. In our benchmark suite, we evaluate it only on AppWorld, the only benchmark that provides a training split. **ACE-Online** [41] constructs memory during deployment from user tasks as they arrive. This removes the need for a pre-collected task set, but adds deployment-time memory-construction cost and latency, and the agent begins with empty memory, exposing early failures to users during the cold-start period.

Implementation Details. We use DeepSeek-V3.2 [3] without reasoning mode as the base LLM for all components, including the Proposer, Solver, Validator, and memory-update calls in PREPING. The same model is also used across all methods for task execution and (when applicable) memory construction. The main results are averaged over three independent construction-and-evaluation runs. We use temperature 0 for Solver execution, 0.7 for Validator judgments and memory updates, and 1.0 for synthetic task proposal. The Validator assigns 1-5 Likert scores for task feasibility and task completion. Trajectories are admitted into solver memory at feasibility scores of 5 for AppWorld and MCP-Universe and at least 2 for BFCL v3, while completion scores of at least 4 indicate successful execution. Each PREPING construction run uses 10 iterations with 10 synthetic tasks per iteration, yielding 100 synthetic practice tasks in total. To preserve the pre-task setting, pre-task methods may use only environment documentation $\mathcal{D}$, such as tool descriptions or API documentation, and feedback obtained from self-generated interactions during memory construction. Additional implementation details, including the full construction algorithm and prompt templates, are provided in Sec. A.

Table 2: **Component ablation.** Score uses AppWorld Test-Normal TGC/SGC and BFCL v3 Base success rate. Env. Info denotes environment information. Naive Task Generation disables all components; PREPING enables all. Bold and underline mark the best and second-best values.

Benchmark	Validator	Env. Info	Practice History	Score ↑	Infeasible Task (%) ↓	Unique Tools ↑	Tool Entropy ↑	W. Recall ↑
AppWorld	✗	✗	✗	47.8 / 26.8	–	65.7	5.611	0.653
	✓	✗	✗	78.2 / 60.7	26.3	69.0	5.586	0.632
	✓	✗	✓	<u>81.5</u> / <u>67.9</u>	33.3	<u>81.7</u>	<u>5.775</u>	<u>0.674</u>
	✓	✓	✗	80.3 / 64.9	**5.3**	42.3	4.711	0.368
	✓	✓	✓	**83.7 / 70.2**	<u>16.0</u>	**87.0**	**5.919**	**0.703**
BFCL v3	✗	✗	✗	59.5	–	47.0	5.022	0.526
	✓	✗	✗	62.0	9.0	44.7	5.015	0.505
	✓	✗	✓	60.8	19.0	**118.0**	<u>6.028</u>	**0.968**
	✓	✓	✗	<u>64.0</u>	<u>6.0</u>	50.3	5.022	0.591
	✓	✓	✓	**65.2**	**4.0**	<u>105.0</u>	**6.095**	<u>0.846</u>

4.2 Experimental Results and Analyses

PREPING Builds Strong Memory Without Target-Task Experience. Table 1 shows that pre-task methods based on documentation or free-form exploration provide limited gains. Direct Memory converts documentation into memory without observing tool execution, which limits its ability to capture procedural knowledge about tool behavior, state change, and failure conditions. Random and Guided Exploration benefit from execution feedback (especially on BFCL v3), but their interaction is not organized around task-level objectives, often leading to shallow, repetitive, or weakly compositional tool use (rather than reusable workflows involving multi-step tool composition). In contrast, PREPING constructs memory from proposer-guided synthetic tasks and validation-filtered trajectories, achieving the strongest pre-task performance across all three benchmarks and improving average score over Base by 17.1 points on AppWorld, 19.3 points on BFCL v3, and 5.4 points on MCP-Universe. Furthermore, despite using no human-defined or deployment-time user tasks for memory construction, PREPING remains competitive with task-informed methods. Specifically, on AppWorld, PREPING exceeds ACE-Offline and is close to ACE-Online, while on BFCL v3, it surpasses ACE-Online on average. These results suggest that PREPING can produce reusable procedural memory that supports downstream task solving even before any target-task experience is available.

Ablation Study: Controlled and Validated Practice Drives the Gains. We next ask which parts of PREPING are responsible for the gains. Table 2 ablates three components: validation-gated memory admission, practice history, and environment information. The all-disabled rows correspond to Naive Task Generation, while the full rows enable all three components. To analyze deeply beyond final task performance, we also report four construction-side diagnostics: infeasible task rate, unique tools, tool entropy, and weighted recall over tools used in deployment-time test trajectories. Infeasible task rate measures how often the validator rejects generated tasks as infeasible. Unique tools counts distinct AppWorld domain APIs or BFCL functions invoked in synthetic trajectories, and tool entropy measures how evenly practice is distributed across them. Weighted recall (W. Recall) compares the tools covered during construction with those used in deployment-time test trajectories, weighting each covered tool by its test-time frequency. Together, these metrics separate three desiderata for pre-task memory construction: task quality, support expansion, and relevance to downstream workflows.

Table 2 first shows that validation-gated admission is crucial for constructing reusable memory. On AppWorld, adding Validator to Naive Task Generation improves performance from 47.8/26.8 to 78.2/60.7. This gain cannot be explained by tool breadth alone: Naive already invokes 65.7 unique APIs with a weighted recall of 0.653, but its unfiltered trajectories can distill infeasible objectives, constraint violations, or failed recovery patterns into solver memory. Sec. B.7 illustrates this failure mode. On BFCL v3, validation also improves average performance from 59.5 to 62.0, although the gain is smaller because infeasible task rates are lower in this setting.

Beyond validation, proposer memory determines which parts of the environment are practiced. Practice history primarily expands support by discouraging repeated practice and pushing proposals toward under-covered tools, increasing AppWorld unique APIs from 69.0 to 81.7 and BFCL v3 from 44.7 to 118.0. Yet, history alone can over-expand into poorly grounded proposals because it tracks what has been practiced but not necessarily what is executable in the current environment. Environmental

Table 3: **PREPING warm-starts online memory construction.** PREPING+ACE initializes ACE-Online with pre-task memory and applies the standard online update procedure.

	AppWorld					BFCL v3		
Method	N-TGC	N-SGC	C-TGC	C-SGC	Avg.	Base	Ctx.	Avg.
Pre-Task Methods								
Base	69.6	49.4	56.7	36.7	53.1	43.7	40.0	41.9
PREPING	83.7	70.2	72.2	54.7	70.2	65.2	59.3	62.3
Task-Informed Methods								
ACE-Online	80.4	65.5	78.3	60.9	71.3	62.3	55.0	58.7
PREPING+ACE	**86.1**	**73.8**	**80.1**	**65.2**	**76.3**	**66.0**	**63.7**	**64.9**

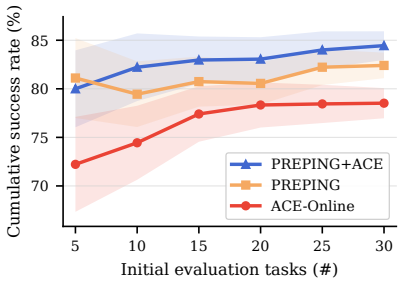

Figure 3: **PREPING mitigates online cold-start before memory builds up.**

information has a complementary effect: it keeps proposals anchored to observed entities, states, and constraints, but by itself lacks the history needed to broaden practice. PREPING combines both signals, yielding the best downstream performance on both benchmarks while maintaining strong tool entropy and weighted recall. These results show that effective pre-task memory construction requires more than generating synthetic tasks at scale; it requires balancing feasibility, coverage expansion, and relevance to workflows the solver will later encounter. We provide iteration-wise construction dynamics for these ablations in Sec. B.2, further decompose the Validator's role in Sec. B.3, and present an AppWorld example of this proposer-memory interaction in Sec. B.6.

PREPING+ACE Improves Online Learning Through Better Initialization. PREPING can also serve as an initialization for online memory construction. Before deployment, it constructs reusable solver memory from synthetic practice without target-environment tasks. During deployment, this memory can then continue to be updated as real user tasks arrive. We call this setting PREPING+ACE, initializing ACE-Online with PREPING memory and then running the standard online update procedure during evaluation. As shown in Table 3, PREPING+ACE improves AppWorld average performance from 71.3 to 76.3. The improvement is especially clear on AppWorld Test-Challenge, where it surpasses ACE-Online on both TGC and SGC, increasing TGC from 78.3 to 80.1 and SGC from 60.9 to 65.2. On the evaluated BFCL v3 categories, the same initialization improves Base performance from 62.3 to 66.0 and Long Context performance from 55.0 to 63.7, raising their average from 58.7 to 64.9. These results show that PREPING is not only useful as a frozen deployment artifact, but also as a strong starting point for online memory construction.

PREPING Reduces Early Online Cold-Start Failures. Fig. 3 examines the early deployment regime, before ACE-Online has accumulated sufficient task-informed memory. We construct 18 shuffled 30-task streams from AppWorld Test-Normal and measure cumulative success over the first k tasks in each stream. Then, ACE-Online shows a clear cold start: its first 10-task success rate is 74.4%, much closer to the no-memory Base at 69.6% than to its full Test-Normal TGC of 80.4%. In contrast, PREPING starts with pre-task memory, and PREPING+ACE further updates it online, achieving 79.4% and 82.2% over the first 10 tasks, respectively. This advantage persists through the full 30-task stream, showing that PREPING and PREPING+ACE mitigate online cold start failures by giving the agent useful procedural memory before substantial user-task experience is available.

PREPING Reduces Tool-Coverage Cold Start. We next examine whether early performance gaps reflect limited tool coverage in online memory. The right panel of Fig. 1 shows that ACE-Online starts with empty memory and only accumulates coverage from evaluation tasks, while PREPING enters deployment with coverage from 100 synthetic practice tasks, and PREPING+ACE starts from this and further expands it with user-task updates. On AppWorld Test-Normal, ACE-Online requires 58 evaluation tasks to match PREPING's pre-deployment coverage; on BFCL v3 Base, it still falls short after all 200 tasks. This shows that synthetic practice reduces the coverage lag of online memory, and that PREPING+ACE benefits by combining broad pre-task coverage with task-informed evidence.

Generalization Across Backbone Models. We further test whether PREPING transfers across backbone models rather than relying on a single primary LLM. On AppWorld Test-Normal, we run the same, full construction-and-evaluation pipeline with GPT-5.1 [18], GPT-OSS-120B [19], and Qwen3-235B-A22B [29], with reasoning-disabled modes for GPT-5.1 and Qwen3-235B-A22B. For each run,

Table 4: **PREPING generalizes across backbone models.** Results are reported on AppWorld Test-Normal with each method run using a single backbone model.

Method	GPT-5.1		GPT-OSS-120B		Qwen3-235B-A22B	
	TGC	SGC	TGC	SGC	TGC	SGC
Pre-Task Methods						
Base	52.4	30.4	28.0	7.1	58.3	37.5
PREPING	**57.7**	**41.1**	**36.9**	**17.9**	**67.3**	**44.6**
Task-Informed Methods						
ACE-Offline	56.0	37.5	36.3	16.1	69.6	50.0
ACE-Online	52.4	33.9	31.5	12.5	66.1	44.6

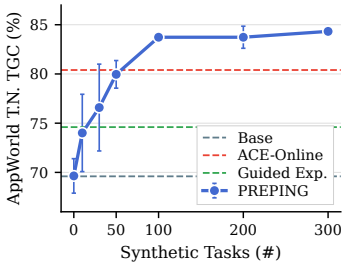

Figure 4: **Construction budget.** AppWorld Test-Normal TGC as a function of the number of synthetic tasks.

the same backbone instantiates the Proposer, Solver, Validator, and memory-update calls, matching the single-model setup used in our main experiments. As shown in Table 4, PREPING improves over the no-memory Base across all three backbones and remains comparable to task-informed methods, despite using no human-defined tasks during construction. The improvement also holds for GPT-OSS-120B, the relatively weaker backbone in this comparison, indicating that PREPING can benefit agents even when the base solver is less capable. Overall, these results suggest that the benefit comes from the controlled memory-construction procedure rather than a model-specific artifact.

Modest Synthetic-Task Budgets Already Yield Strong Gains. To understand how many synthetic tasks are needed to construct effective memory, we vary the construction budget by the number of synthetic tasks. As shown in Fig. 4, PREPING improves steadily as the construction progresses, reaching 76.6 after only 30 synthetic tasks and already surpassing Guided Exploration. With 50 synthetic tasks, it reaches 80.0, approaching the ACE-Online reference at 80.6. This suggests that proposer-guided practice can expose broad and useful procedures even under modest construction budgets. Additional practice continues to improve performance, with 300 synthetic tasks reaching 84.3, albeit with smaller marginal gains. Thus, the 100-task budget used in our main experiments provides a practical balance between construction cost and performance gain.

Task-Seeded PREPING Guides Synthetic Practice. Although constructing a large, diverse offline task set can be costly, a small seed set may be available before deployment. We therefore evaluate Task-Seeded PREPING, which starts from 10 solved tasks (sampled from the AppWorld training split) and then runs the same iterative PREPING loop as in the main setting. It initializes proposer memory from the sampled task instructions and solver memory from their corresponding trajectories. As shown in Table 5, Task-Seeded PREPING improves AppWorld Test-Normal TGC/SGC from 83.7/70.2 to 85.1/73.8. The result suggests that PREPING can leverage a small offline set as an initial anchor for synthetic task proposal, while still expanding solver memory through subsequent synthetic practice.

Table 5: **PREPING with a 10 offline task set.** Results are averaged over three runs.

Method	TGC	SGC
Base	69.6	49.4
PREPING	83.7	70.2
Task-Seeded PREPING	**85.1**	**73.8**

PREPING Reduces Deployment-Time Cost. Online memory construction can improve performance, but it also adds memory-update calls during user-facing deployment. In contrast, PREPING constructs solver memory before deployment, allowing user tasks to be executed without additional memory-construction calls. As shown in Fig. 5, this substantially reduces deployment-time cost per task, with ACE-Online costing about 2.99× more on AppWorld and 2.23× more on BFCL v3. This comparison isolates deployment-time cost from the one-time pre-deployment construction cost of PREPING. Even when this construction cost is included, PREPING remains cheaper than ACE-Online on both AppWorld and BFCL v3, as shown in Table 14. Thus, PREPING shifts memory construction from a recurring deployment-time expense to an amortizable pre-deployment investment.

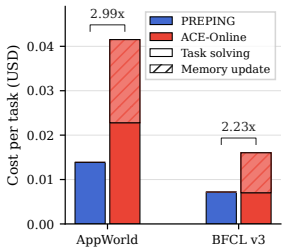

Figure 5: **Deployment time cost per task.**

5 Conclusion

We studied *pre-task memory construction*, the problem of building reusable agent memory before any target-environment task data are available. PREPING addresses this setting by treating memory construction as a control problem over what to practice and what to store: proposer memory shapes synthetic practice, the solver executes it in the target environment, and the validator filters out infeasible trajectories before solver-memory updates. Experiments across AppWorld, BFCL v3, and MCP-Universe show that procedural memory can be constructed before deployment, outperforming pre-task baselines and providing a strong initialization for online memory adaptation. These results suggest that agent memory need not be accumulated only passively from deployment-time interactions; it can also be actively prepared through controlled, environment-grounded practice before deployment. We discuss limitations and broader impact in Sec. C.

Acknowledgements

This work was supported by Institute for Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (RS-2019-II190075, Artificial Intelligence Graduate School Program (KAIST); RS-2022-II220713, Meta-learning Applicable to Real-world Problems; RS-2024-00469482 & RS-2024-00509279, Global AI Frontier Lab International Collaborative Research), and by National Research Foundation of Korea (NRF) grants funded by the Korea government (MSIT) (RS-2023-00256259; RS-2026-25488933). Support was also provided by the InnoCORE program of the Ministry of Science and ICT (No. N10250156) and the "Advanced GPU Utilization Support Program" funded by the Government of the Republic of Korea (Ministry of Science and ICT).

References

[1] Emre Can Acikgoz, Cheng Qian, Jonas Hübotter, Heng Ji, Dilek Hakkani-Tür, and Gokhan Tur. Tool-R0: Self-evolving LLM agents for tool-learning from zero data. *arXiv*, 2026. doi: 10.48550/arXiv.2602.21320.

[2] Anthropic. Introducing the model context protocol, November 2024. URL https://www.anthropic.com/news/model-context-protocol. Accessed: 2026-05-01.

[3] DeepSeek-AI. DeepSeek-V3.2: Pushing the frontier of open large language models, 2025. URL https://huggingface.co/deepseek-ai/DeepSeek-V3.2. Model card.

[4] Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Qihan Ren, Yiran Wu, Hongru Wang, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026.

[5] Dongge Han, Camille Couturier, Daniel Madrigal Díaz, Xuchao Zhang, Victor Rühle, and Saravan Rajmohan. Legomem: Modular procedural memory for multi-agent LLM systems for workflow automation. *arXiv*, 2025. doi: 10.48550/arXiv.2510.04851.

[6] Mengkang Hu, Pu Zhao, Can Xu, Qingfeng Sun, Jian-Guang Lou, Qingwei Lin, Ping Luo, and Saravan Rajmohan. AgentGen: Enhancing planning abilities for large language model based agent via environment and task generation. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.1*, pages 496–507, 2025. doi: 10.1145/3690624.3709321.

[7] Yuyang Hu, Shichun Liu, Yanwei Yue, Guibin Zhang, Boyang Liu, Fangyi Zhu, Jiahang Lin, Honglin Guo, Shihan Dou, Zhiheng Xi, Senjie Jin, Jiejun Tan, Yanbin Yin, Jiongnan Liu, Zeyu Zhang, Zhongxiang Sun, Yutao Zhu, Hao Sun, Boci Peng, Zhenrong Cheng, Xuanbo Fan, Jiaxin Guo, Xinlei Yu, Zhenhong Zhou, Zewen Hu, Jiahao Huo, Junhao Wang, Yuwei Niu, Yu Wang, Zhenfei Yin, Xiaobin Hu, Yue Liao, Qiankun Li, Kun Wang, Wangchunshu Zhou, Yixin Liu, Dawei Cheng, Qi Zhang, Tao Gui, Shirui Pan, Yan Zhang, Philip Torr, Zhicheng Dou, Ji-Rong

Wen, Xuanjing Huang, Yu-Gang Jiang, and Shuicheng Yan. Memory in the age of AI agents. *arXiv*, 2025. doi: 10.48550/arXiv.2512.13564.

[8] Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning LLM from zero data. *arXiv*, 2025. doi: 10.48550/arXiv.2508.05004.

[9] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. In *International Conference on Learning Representations (ICLR)*, 2023.

[10] Bo Liu, Chuanyang Jin, Seungone Kim, Weizhe Yuan, Wenting Zhao, Ilia Kulikov, Xian Li, Sainbayar Sukhbaatar, Jack Lanchantin, and Jason Weston. Spice: Self-play in corpus environments improves reasoning. *arXiv*, 2025. doi: 10.48550/arXiv.2510.24684.

[11] Jinghao Luo, Yuchen Tian, Chuxue Cao, Ziyang Luo, Hongzhan Lin, Kaixin Li, Chuyi Kong, Ruichao Yang, and Jing Ma. From storage to experience: A survey on the evolution of LLM agent memory mechanisms. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 41622–41652, 2026. doi: 10.18653/v1/2026.findings-acl.2069.

[12] Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. MCP-universe: Benchmarking large language models with real-world model context protocol servers. *arXiv*, 2025. doi: 10.48550/arXiv.2508.14704.

[13] Bodhisattwa Prasad Majumder, Bhavana Dalvi Mishra, Peter Jansen, Oyvind Tafjord, Niket Tandon, Li Zhang, Chris Callison-Burch, and Peter Clark. CLIN: A continually learning language agent for rapid task adaptation and generalization. *arXiv*, 2023. doi: 10.48550/arXiv.2310.10134.

[14] Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, Chenlin Zhou, Jiayi Mao, Tianze Xia, Jiafeng Guo, and Shenghua Liu. A survey of context engineering for large language models. *arXiv*, 2025. doi: 10.48550/arXiv.2507.13334.

[15] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv*, 2026. doi: 10.48550/arXiv.2601.11868.

[16] Shikhar Murty, Hao Zhu, Dzmitry Bahdanau, and Christopher D. Manning. NNetNav: Unsupervised learning of browser agents through environment interaction in the wild. *arXiv*, 2024. doi: 10.48550/arXiv.2410.02907.

[17] Hadi Nekoei, Aman Jaiswal, Patrice Béchard, Oleh Shliazhko, Orlando Marquez Ayala, Mathieu Reymond, Massimo Caccia, Alexandre Drouin, Sarath Chandar, and Alexandre Lacoste. JEF-hinter: Leveraging offline knowledge for improving web agents adaptation. *arXiv*, 2025. doi: 10.48550/arXiv.2510.04373.

[18] OpenAI. GPT-5.1 model, 2025. URL https://platform.openai.com/docs/models/gpt-5.1/. OpenAI API documentation; accessed: 2026-05-02.

[19] OpenAI. gpt-oss-120b and gpt-oss-20b model card, August 2025. URL https://openai.com/index/gpt-oss-model-card/. Accessed: 2026-05-02.

[20] 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. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations (ICLR)*, 2026.

[21] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv*, 2023. doi: 10.48550/arXiv.2310.08560.

[22] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023. doi: 10.1145/3586183.3606763.

[23] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Neural Information Processing Systems (NeurIPS)*, 2024.

[24] Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *International Conference on Machine Learning (ICML)*, 2025.

[25] 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. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations (ICLR)*, 2024.

[26] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Neural Information Processing Systems (NeurIPS)*, 2023.

[27] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Neural Information Processing Systems (NeurIPS)*, 2023. doi: 10.52202/075280-0377.

[28] Mirac Suzgun, Mert Yüksekgönul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. *arXiv*, 2025. doi: 10.48550/arXiv.2504.07952.

[29] Qwen Team. Qwen3 technical report. *arXiv*, 2025. doi: 10.48550/arXiv.2505.09388.

[30] Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 16022–16076, 2024. doi: 10.18653/v1/2024.acl-long.850.

[31] Zilong Wang, Yuedong Cui, Li Zhong, Zimin Zhang, Da Yin, Bill Yuchen Lin, and Jingbo Shang. OfficeBench: Benchmarking language agents across multiple applications for office automation. *arXiv*, 2024. doi: 10.48550/arXiv.2407.19056.

[32] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *International Conference on Machine Learning (ICML)*, 2025.

[33] Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida I. Wang. Toward training superintelligent software agents through self-play SWE-RL. *arXiv*, 2025. doi: 10.48550/arXiv.2512.18552.

[34] Peng Xia, Kaide Zeng, Jiaqi Liu, Can Qin, Fang Wu, Yiyang Zhou, Caiming Xiong, and Huaxiu Yao. Agent0: Unleashing self-evolving agents from zero data via tool-integrated reasoning. *arXiv*, 2025. doi: 10.48550/arXiv.2511.16043.

[35] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic memory for LLM agents. In *Neural Information Processing Systems (NeurIPS)*, 2025. doi: 10.52202/085713-0593.

[36] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[37] Zhenrui Yue, Kartikeya Upasani, Xianjun Yang, Suyu Ge, Shaoliang Nie, Yuning Mao, Zhe Liu, and Dong Wang. Dr. zero: Self-evolving search agents without training data. *arXiv*, 2026. doi: 10.48550/arXiv.2601.07055.

[38] Yunpeng Zhai, Shuchang Tao, Cheng Chen, Anni Zou, Ziqian Chen, Qingxu Fu, Shinji Mai, Li Yu, Jiaji Deng, Zouying Cao, Zhaoyang Liu, Bolin Ding, and Jingren Zhou. Agentevolver: Towards efficient self-evolving agent system. *arXiv*, 2025. doi: 10.48550/arXiv.2511.10395.

[39] Guibin Zhang, Muxin Fu, Guancheng Wan, Miao Yu, Kun Wang, and Shuicheng Yan. G-memory: Tracing hierarchical memory for multi-agent systems. *arXiv*, 2025. doi: 10.48550/arXiv.2506.07398.

[40] Guibin Zhang, Haotian Ren, Chong Zhan, Zhenhong Zhou, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. Memevolve: Meta-evolution of agent memory systems. *arXiv*, 2025. doi: 10.48550/arXiv.2512.18746.

[41] 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. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations (ICLR)*, 2026.

[42] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936.

[43] Andrew Zhao, Yiran Wu, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In *Neural Information Processing Systems (NeurIPS)*, 2025. doi: 10.52202/085713-3534.

[44] Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *International Conference on Learning Representations (ICLR)*, 2024.

[45] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 19724–19731, 2024. doi: 10.1609/aaai.v38i17.29946.

[46] Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, and Jun Wang. Memento: Fine-tuning LLM agents without fine-tuning LLMs. *arXiv*, 2025. doi: 10.48550/arXiv.2508.16153.

[47] Yifei Zhou, Sergey Levine, Jason E. Weston, Xian Li, and Sainbayar Sukhbaatar. Self-challenging language model agents. In *Neural Information Processing Systems (NeurIPS)*, 2025.

[48] Yifei Zhou, Qianlan Yang, Kaixiang Lin, Min Bai, Xiong Zhou, Yu-Xiong Wang, Sergey Levine, and Li Erran Li. Proposer-agent-evaluator (PAE): Autonomous skill discovery for foundation model internet agents. In *International Conference on Machine Learning (ICML)*, pages 79490–79528, 2025.

Appendix

The appendix complements the main text by recording the implementation details needed to reproduce PREPING and providing additional analyses that help interpret the main results.

- **Additional Experimental Details** (Sec. A).
  - Full Algorithm of PREPING (Sec. A.1).
  - Meta Prompts to Implement PREPING (Sec. A.2).
  - ACE Reflector-Curator Memory-Induction Prompts (Sec. A.3).
  - Task-Solving Prompts (Sec. A.4).
  - Details on Baselines (Sec. A.5).
  - Compute Resources (Sec. A.6).
- **Additional Experimental Results and Analysis** (Sec. B).
  - Full Main Results with Standard Deviations (Sec. B.1).
  - Iteration Dynamics of Component Ablations (Sec. B.2).
  - Ablating Validator Signals in Memory Updates (Sec. B.3).
  - Validator Reliability Audit (Sec. B.4).
  - Robustness to Incomplete and Noisy Documentation (Sec. B.5).
  - Qualitative Example of AppWorld Proposer Memory (Sec. B.6).
  - Qualitative Example of Infeasible Tasks and Memory Contamination (Sec. B.7).
  - Trajectory Steps in Synthetic Practice and Online Evaluation (Sec. B.8).
  - Construction-Inclusive Cost (Sec. B.9).
- **Limitations and Broader Impact** (Sec. C).

A Additional Experimental Details

A.1 Full Algorithm of PREPING

Algorithm 1 PREPING: Pre-Task Memory Construction

Require: Target executable environment E, Tool documentation $\mathcal{D}$, Construction iterations T, Batch size N

Ensure: Solver memory M_{sol}

$M_{\text{prop}} \leftarrow \varnothing, \quad M_{\text{sol}} \leftarrow \varnothing$
for $t = 1, \ldots, T$ **do**
  $\mathcal{X}_t = x_t^{1:N} = \{x_t^i\}_{i=1}^N \leftarrow A_{\text{prop}}(M_{\text{prop}}, \mathcal{D}; N)$ ▷ task generation; Fig. 6
  for $i = 1, \ldots, N$ **do**
    $\tau_t^i \leftarrow A_{\text{sol}}(x_t^i, M_{\text{sol}}, E)$ ▷ task execution; Sec. A.4
    $v_t^i \leftarrow A_{\text{val}}(x_t^i, \tau_t^i)$ ▷ trajectory validation; Fig. 7
    $M_{\text{prop}} \leftarrow U_{\text{prop}}(M_{\text{prop}}, x_t^i, \tau_t^i, v_t^i)$ ▷ update task history and grounded env info; Fig. 8
    if Feasible(v_t^i) **then**
      $M_{\text{sol}} \leftarrow U_{\text{sol}}(M_{\text{sol}}, x_t^i, \tau_t^i, v_t^i)$ ▷ solver-memory update; Sec. A.3
    end if
  end for
end for
return M_{sol}

A.2 Meta Prompts to Implement PREPING

In this section, we provide the BFCL instantiation of the meta prompts used for synthetic task generation, grounded environment information summarization, and trajectory validation in PREPING.

```
You are a synthetic BFCL tool-use task generator.

Your goal is to generate diverse, realistic, and challenging task instructions that an AI assistant would need to
complete.

Server function docs:
{api_docs}

## Guidelines
- Feasibility: Generate tasks that are actually executable with the given functions
- Each task must be solvable in one user request.
- Ground tasks in the available server docs and any provided environment information.
- If environment information is provided, prefer entities, files, users, tickets, airports, and symbols that are
explicitly supported there.
- If environment information is missing or sparse, avoid inventing highly specific filenames, IDs, or credentials
unless they are already implied by the server docs or prior task history.
- Prefer tasks that teach reusable agent behavior: inspect state, retrieve the right information, transform it,
then act.
- Avoid vague requests such as "look around", "analyze everything", or tasks without a clear finish condition.
- Only require a user-facing final answer when an explicit output action is relevant, such as posting, messaging,
commenting, writing, logging, or displaying through a tool.
- Otherwise, frame the task around the required tool or state operation, not around "telling", "showing", or "
reporting" the answer.
- Do not infer an output requirement from user-facing wording alone.
- Use only servers that are relevant to the task.
- Do not assume hidden files, extra tickets, custom credentials, or custom log files unless they were grounded by
environment information.
- ‘intended_functions‘ is optional and is only a lightweight hint for analysis/logging.
- The actual executable function set is determined by the selected ‘servers‘, not by ‘intended_functions‘.

## Environment Information
{environment_info_section}

## Prior Task History
- Use prior task history as weak guidance for where to explore next, not as templates to imitate.
- Push beyond solved tasks with meaningfully harder or more diverse task structures.
- Use failure tasks to understand current model limitations and common mistakes, not as targets to reproduce.
- Use infeasible tasks only to avoid invalid setups or impossible assumptions.
- Treat failure or infeasibility reasons as short hints about missing skills or bad assumptions, not as
instructions to retry the same pattern.
- Avoid near-duplicates: do not merely rename entities or tweak dates, numbers, thresholds, or output format while
keeping the same task pattern.
- Keep the batch diverse across different servers, apps, tools, APIs, entities, reasoning patterns, and action
structures.
- Use the usage statistics as weak context about prior coverage, not as hard constraints or item-specific targets.
- Avoid overcommitting to a narrow set of heavily repeated items, but do not force tasks around specific low-count
items just because they appear less used.
- Prefer feasible expansions that broaden coverage naturally instead of optimizing for any single app, API, server,
 or function name.

{task_history_section}

## Output Format
Return a JSON array of task instructions:
‘‘‘json
[
  {
    "servers": ["Server1", "Server2"],
    "intended_functions": ["Server1.function1", "Server2.function2"],
    "question": "single user request"
  }
]

Generate {num_tasks} diverse task instructions:
```

Figure 6: BFCL task-generation meta prompt used in our implementation of PREPING.

```
You are validating whether a synthetic BFCL task execution produced a useful memory signal.

Evaluate the following Task and Trajectory using a 5-point Likert scale for each criterion.

## Evaluation Criteria

### 1. Task Feasibility
Evaluate whether the task is executable with the available tools and whether every required intermediate fact is
grounded in the trajectory evidence.

- Verify that every required target entity, field, metric, or intermediate value in the instruction is positively
supported by tool outputs as existing and usable for the requested operation.
- A required file, user, ticket, route, credential, or other entity is grounded only if the trajectory provides
positive evidence that it exists and is usable for the requested operation.
- Negative evidence such as "not found", empty matches, missing records, authentication failure, unavailable routes
, or lookup errors should count against feasibility rather than as grounding.
- Failed lookup attempts do not make a required entity grounded.

- **5 (Excellent)**: Fully executable as stated; all required entities, metrics, and intermediate values are
explicitly grounded in trajectory evidence.
- **4 (Good)**: Executable with minor ambiguity, but the needed values and entities are still reasonably supported
by the trajectory.
- **3 (Acceptable)**: Plausibly executable, but one key field, metric, or grounding step is weakly supported or
only partially observed.
- **2 (Poor)**: Likely infeasible in practice; a required entity, field, or intermediate value is missing from
observable tool outputs.
- **1 (Unacceptable)**: Infeasible/contradictory; the task relies on unsupported tools, impossible preconditions,
or missing critical values that cannot be grounded.

### 2. Task Completion
Judge whether the task instruction was successfully completed.

- Evaluate whether all required steps and constraints are satisfied based on the trajectory.

- **5 (Excellent)**: Every requirement and constraint is satisfied.
- **4 (Good)**: The task appears completed with only minor ambiguity, and no important requirement is missing.
- **3 (Acceptable)**: Some progress, but at least one requirement or constraint is missing or weakly supported.
- **2 (Poor)**: Minor progress only; the main outcome is not achieved or remains unverified.
- **1 (Unacceptable)**: No meaningful completion.

Task:
{task_instruction}

Trajectory:
{trajectory}

Return strict JSON with:
{
  "feasibility_reason": "short reason",
  "feasibility_score": 1-5,
  "task_completion_reason": "short reason",
  "task_completion_score": 1-5,
}
```

Figure 7: BFCL validator meta prompt used in our implementation of PREPING.

```
You are analyzing an AI agent's execution trajectory to extract grounded environment observations for future
synthetic task generation.

Your goal is to summarize only reusable environment facts observed in this trajectory.

Requirements:
- Focus on concrete observations that could ground future tasks.
- Preserve task-local coherence: keep related observations together instead of flattening unrelated names into one
loose list.
- Exclude anything created, posted, added, invited, updated, renamed, or otherwise changed by the agent during this
 task unless it reveals an important constraint or action pattern.
- Exclude generic API documentation, schemas, and error messages.
- Prefer observations grounded in BFCL server state, tool outputs, state constraints, and repeated action patterns.
- Keep the summary compact: 2-5 bullet lines total.

Return ONLY JSON:
‘‘‘json
{
  "summary": "- observation 1\n- observation 2"
}
‘‘‘
Task:
{task_instruction}

Trajectory:
{trajectory}
```

Figure 8: BFCL grounded environment information summarization prompt used to extract proposer-side environment information from synthetic trajectories.

A.3 ACE Reflector-Curator Memory-Induction Prompts

We instantiate solver-memory updates with the ACE reflector-curator pipeline. The prompt text closely follows the original ACE reflector-curator prompts [41]. The `{ground_truth_result}` contains benchmark ground-truth feedback for ACE-Offline, the validator output for PREPING, and `None` for ACE-Online. The BFCL and MCP-Universe variants follow the same input-output structure, with environment-specific terminology changed to function calling and MCP-tool execution.

AppWorld ACE Reflector

```
You are an expert AppWorld coding agent and educator. Your job is to diagnose the current trajectory: identify what
 went wrong (or could be better), API usage, and ground truth when applicable.

Instructions:
- Carefully analyze the model's reasoning trace to identify where it went wrong
- Take the environment feedback into account, comparing the predicted answer with the (optional) ground truth to
understand the gap
- Identify specific conceptual errors, calculation mistakes, or misapplied strategies
- Provide actionable insights that could help the model avoid this mistake in the future
- Identify root causes: wrong source of truth, bad filters (timeframe/direction/identity), formatting issues, or
missing authentication and how to correct them.
- Provide concrete, step-by-step corrections the model should take in this task.
- Be specific about what the model should have done differently
- You will receive bulletpoints that are part of playbook that's used by the generator to answer the question.
- You need to analyze these bulletpoints, and give the tag for each bulletpoint, tag can be ['helpful', 'harmful',
'neutral'] (for the generator to generate the correct answer)
- Explicitly curate from the environment feedback the output format/schema of APIs used when unclear or mismatched
with expectations (e.g., apis.blah.show_contents() returns a list of content_ids (strings), not content objects)

Inputs:

Task Instruction:
{{task_instruction}}

Ground Truth Result:
{{ground_truth_result}}

Current Playbook:
{{playbook}}

Agent-Environment Trajectory:
{{trajectory}}

Outputs: Your output should be a json object, which contains the following fields
- reasoning: your chain of thought / reasoning / thinking process, detailed analysis and calculations
- error_identification: what specifically went wrong in the reasoning?
- root_cause_analysis: why did this error occur? What concept was misunderstood?
- correct_approach: what should the model have done instead?
- key_insight: what strategy, formula, or principle should be remembered to avoid this error?
- bullet_tags: a dictionary mapping each bullet_id (the {section}-{number} prefix shown in the playbook) to its tag
 ('helpful', 'harmful', or 'neutral')
Answer in this exact JSON format:

{
"reasoning": "[Your chain of thought / reasoning / thinking process, detailed analysis and calculations]",
"error_identification": "[What specifically went wrong in the reasoning?]",
"root_cause_analysis": "[Why did this error occur? What concept was misunderstood?]",
"correct_approach": "[What should the model have done instead?]",
"key_insight": "[What strategy, formula, or principle should be remembered to avoid this error?]",
"bullet_tags": {"bullet_id_1": "helpful", "bullet_id_2": "harmful", "bullet_id_3": "neutral"}
}
```

Figure 9: AppWorld ACE reflector prompt used for trajectory analysis before playbook update.

AppWorld ACE Curator

```
You are a master curator of knowledge. Your job is to identify what new insights should be added to an existing
playbook based on a reflection from a previous attempt.

Context:
- The playbook you created will be used to help answering similar questions.

Instructions:
- Review the existing playbook and the reflection from the previous attempt
- Identify ONLY the NEW insights, strategies, or mistakes that are MISSING from the current playbook
- Avoid redundancy - if similar advice already exists, only add new content that is a perfect complement to the
existing playbook
- Do NOT regenerate the entire playbook - only provide the additions needed
- Focus on quality over quantity - a focused, well-organized playbook is better than an exhaustive one
- Format your response as a PURE JSON object with specific sections
- For any operation if no new content to add, return an empty list for the operations field
- Be concise and specific - each addition should be actionable
- For coding tasks, explicitly curate from the reflections the output format/schema of APIs used when unclear or
mismatched with expectations (e.g., apis.blah.show_contents() returns a list of content_ids (strings), not content
objects)

Inputs:

Task Instruction:
{{task_instruction}}

Current Playbook:
{{playbook}}

Agent-Environment Trajectory:
{{trajectory}}

Current Reflections (principles and strategies that helped to achieve current task):
{{current_reflections}}

Your Task:
Output ONLY a valid JSON object with these exact fields:
- reasoning: your chain of thought / reasoning / thinking process, detailed analysis and calculations
- operations: a list of operations to be performed on the playbook
- type: the type of operation to be performed
- section: the section to add the bullet to (one of: strategies, code_snippets, pitfalls)
- content: the new content of the bullet

Available Operations:
1. ADD: Create new bullet points with fresh IDs
- section: the section to add the new bullet to
- content: the new content of the bullet. Note: no need to include the bullet_id in the content like '[ctx-00263]
helpful=1 harmful=0 ::', the bullet_id will be added by the system.

RESPONSE FORMAT - Output ONLY this JSON structure (no markdown, no code blocks):
{{
"reasoning": "...",
"operations": [
   {{ "type": "ADD", "section": "...", "content": "..." }}
]
}}
```

Figure 10: AppWorld ACE curator prompt used to merge reusable memory into the solver memory.

A.4 Task-Solving Prompts

In this section, we provide the benchmark-specific prompts used when the agent solves downstream evaluation tasks. Placeholders denote task-specific fields, tool descriptions, and solver memory.

````
USER:
I am your supervisor and you are a super intelligent AI Assistant whose job is to achieve my day-to-day tasks
completely autonomously.

To do this, you will need to interact with app/s (e.g., spotify, venmo etc) using their associated APIs on my
behalf. For this you will undertake a *multi-step conversation* using a python REPL environment. That is, you will
write the python code and the environment will execute it and show you the result, based on which, you will write
python code for the next step and so on, until you've achieved the goal. This environment will let you interact
with app/s using their associated APIs on my behalf.

Here are three key APIs that you need to know to get more information

# To get a list of apps that are available to you.

```python
print(apis.api_docs.show_app_descriptions())
```

# To get the list of apis under any app listed above, e.g. spotify

```python
print(apis.api_docs.show_api_descriptions(app_name='spotify'))
```

# To get the specification of a particular api, e.g. spotify app's login api

```python
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='login'))
```

Each code execution will produce an output that you can use in subsequent calls. Using these APIs, you can now
generate code, that I will execute, to solve the task.

{{Solver Memory}}

[Fixed AppWorld in-context walkthrough omitted for length.]

**Key instructions**:
(1) Make sure to end code blocks with ``` followed by a newline(\n).
(2) Remember you can use the variables in your code in subsequent code blocks.
(3) Remember that the email addresses, access tokens and variables (e.g. spotify_password) in the example above are
 not valid anymore.
(4) You can use the "supervisor" app to get information about my accounts and use the "phone" app to get
information about friends and family.
(5) Always look at API specifications (using apis.api_docs.show_api_doc) before calling an API.
(6) Write small chunks of code and only one chunk of code in every step. Make sure everything is working correctly
before making any irreversible change.
(7) Many APIs return items in "pages". Make sure to run through all the pages by looping over page_index.
(8) Once you have completed the task, make sure to call apis.supervisor.complete_task(). If the task asked for some
 information, return it as the answer argument, i.e. call apis.supervisor.complete_task(answer=<answer>). Many
tasks do not require an answer, so in those cases, just call apis.supervisor.complete_task() i.e. do not pass any
argument.

Using these APIs, now generate code to solve the actual task:

My name is: {{ supervisor.first_name }} {{ supervisor.last_name }}. My personal email is {{ supervisor.email }} and
 phone number is {{ supervisor.phone_number }}.
Task: {{ instruction }}
````

Figure 11: AppWorld task-solving generation prompt.

```
SYSTEM:
You are an expert in composing functions. You are given a question and a set of possible functions. Based on the
question, you will need to make one or more function/tool calls to achieve the purpose.
If none of the functions can be used, point it out. If the given question lacks the parameters required by the
function, also point it out.
You should only return the function calls in your response.

If you decide to invoke any of the function(s), you MUST put it in the format of
[func_name1(params_name1=params_value1, params_name2=params_value2...), func_name2(params)].
You SHOULD NOT include any other text in the response.

At each turn, you should try your best to complete the tasks requested by the user within the current turn.
Continue to output functions to call until you have fulfilled the user's request to the best of your ability. Once
you have no more functions to call, the system will consider the current turn complete and proceed to the next turn
 or task.

Here is a list of functions in JSON format that you can invoke.
{{functions}}

{{Solver Memory}}

USER:
{BFCL multi-turn user message for the current turn}
```

Figure 12: BFCL task-solving generation prompt.

```
USER:
You are a ReAct (Reasoning and Acting) agent.
{{INSTRUCTION}}

You have access to these tools:
### Tools Description ###
{{TOOLS_DESCRIPTION}}
### End of Tools Description ###

{{Solver Memory}}

You need to answer the following question:

Question: {{QUESTION}}

Your goal is to reason about the question and decide on the best course of action to answer it accurately.
You need to choose the appropriate tool based on the question. If no tool is needed, reply directly.
Please use only the tools that are explicitly defined above.

Instructions:
1. Analyze the query, previous reasoning steps, and results.
2. Decide on the next action: use a tool or provide a final answer.
3. You MUST output the final answer within {{MAX_STEPS}} steps.
4. Respond in the following JSON format:

If you need to use a tool:
{
    "thought": "Your detailed reasoning about what to do next",
    "action": {
        "reason": "Explanation of why you chose this tool",
        "server": "server-name",
        "tool": "tool-name",
        "arguments": {
            "argument-name": "argument-value"
        }
    }
}

If you have enough information to answer the query:
{
    "thought": "Your final reasoning process to derive the answer.",
    "answer": "Final answer to the query"
}

Remember:
- Be thorough in your reasoning.
- Use tools when you need more information.
- Always base your reasoning on the actual results from tool use.
- If a tool returns no results or fails, acknowledge this and consider using a different tool or approach.
- Provide a final answer when you're confident you have sufficient information.
- The response must be in a valid JSON format.
```

Figure 13: MCP-Universe task-solving generation prompt.

A.5 Details on Baselines

For a fair comparison with PREPING's construction budget of 100 synthetic tasks, the baselines use 100 construction trajectories for memory induction. The exception is AppWorld exploration, where we use one exploration trajectory from each of the 90 train-task environments.

Direct Memory. Direct Memory tests whether environment documentation alone is sufficient to construct useful solver memory. It does not execute synthetic tasks or free-form exploration. Instead, it converts static environment documents (e.g., API documentation, tool descriptions) into synthetic action-output trajectories and feeds those trajectories through the same reflector-curator memory induction pipeline [41] used by the other memory baselines. Specifically, in AppWorld, each trajectory is constructed by sampling a subset of app API documentation. In BFCL, each trajectory is constructed from structured function documentation grouped by server for the target BFCL category. In MCP-Universe, each trajectory is constructed using tools from the MCP servers available for each test category, sampling a subset of servers per iteration. This baseline therefore isolates memory induced from documentation-only environment access, without observing target-task trajectories or exploratory tool outcomes.

Random and guided exploration. Random Exploration and Guided Exploration are task-free memory-construction baselines that execute free-form environment interaction and convert the resulting trajectories into playbook memory. They use the same downstream solver and reflector-curator memory induction pipeline [41], differing only in how exploration trajectories are generated. Random Exploration uses an unconstrained exploration instruction, while Guided Exploration adds lightweight coverage history to prefer tools that have not yet been exercised. In BFCL, each trajectory is generated by a synthetic BFCL task exposing all function documentation available for the benchmark, and the exploration instruction is instantiated as the BFCL user message. We add an instruction limiting each exploration turn to at most three function calls, as otherwise the explorer often attempted to call nearly all available functions in a single turn. We then induce the playbook from action-output trajectories, with Guided Exploration appending cumulative tool coverage to the next task exploration prompt. In AppWorld, each trajectory is generated by running an explorer agent in the Python REPL: the random prompt asks the model to inspect apps, read API documentation, and test diverse endpoints, while the guided prompt additionally includes shared API coverage statistics. Figs. 14 and 15 show the corresponding prompts.

BFCL Random Exploration

```
Explore the environment and available tools by making real tool calls that can
build reusable memory for future tasks. Prefer concise, informative exploration:
make a tool call when it is likely to reveal new behavior, confirm an important
state change, or expose a useful constraint. Avoid brute-force enumeration or
repetitive calls that do not add new information.
```

BFCL Guided Exploration

```
Explore the environment and available tools by making real tool calls that can
build reusable memory for future tasks. Prefer concise, informative exploration:
make a tool call when it is likely to reveal new behavior, confirm an important
state change, or expose a useful constraint. Avoid brute-force enumeration or
repetitive calls that do not add new information.

When natural, prefer learning reusable facts from functions that have not been
explored yet, but do not optimize for raw call volume. Prefer representative
probes before switching domains, and use outputs to decide what to try next.

Already explored functions:
{coverage_summary}
```

Figure 14: **BFCL exploration baseline prompt templates.**

A.6 Compute Resources

All experiments used API-hosted LLM inference, so no local GPU training was performed. Agent execution and memory-construction jobs were run on CPU workers with parallel API calls; the main compute cost is therefore reported in token cost rather than GPU-hours. Wall-clock time varies with provider latency and worker parallelism.

AppWorld Random Exploration

```
You are an AI assistant randomly exploring the AppWorld environment.

Your goal is to DISCOVER and TEST various APIs in a random, exploratory manner.
You have access to a Python REPL environment where you can execute code.

EXPLORATION STRATEGY:
1. Try different apps randomly - don't stick to one app too long.
2. Test various API endpoints with different parameters.
3. Be curious and try unexpected combinations.
4. Observe outputs carefully - note formats, errors, edge cases.
5. When you feel you've explored enough, call: apis.supervisor.complete_task().

Start by checking what apps are available with apis.api_docs.show_app_descriptions().
Before calling an API, inspect its documentation with apis.api_docs.show_api_doc(...).
Write small code chunks, use only one code block per step, and verify behavior before
making irreversible changes. Many APIs return paginated results; loop over page_index
when needed. If a final answer is required, pass it as complete_task(answer=...);
otherwise call complete_task() without an answer.
```

AppWorld Guided Exploration

```
You are an AI assistant systematically exploring the AppWorld environment.

Your goal is to MAXIMIZE COVERAGE by visiting NEW, unexplored APIs.
You have access to a Python REPL environment where you can execute code.

EXPLORATION STRATEGY:
1. PRIORITIZE APIs you haven't visited before.
2. Test various API endpoints with different parameters.
3. Be curious and try unexpected combinations.
4. Observe outputs carefully - note formats, errors, edge cases.
5. When you feel you've explored enough, call: apis.supervisor.complete_task().

=== EXPLORATION PROGRESS ===
Unique APIs visited so far: {unique_apis_visited}
Total API calls made: {total_api_calls}
Apps explored: {apps_explored}

=== ALREADY VISITED APIs ===
{visited_api_summary}

Focus on exploring NEW APIs that are NOT in the list above. Start by checking
available apps and finding APIs you have not tried yet.
```

Shared AppWorld Execution Rules

````
Here are three key APIs that you need to know to get more information:

# To get a list of apps that are available to you.
```python
print(apis.api_docs.show_app_descriptions())
```

# To get the list of APIs under any app listed above, e.g. spotify.
```python
print(apis.api_docs.show_api_descriptions(app_name='spotify'))
```

# To get the specification of a particular API, e.g. spotify app's login API.
```python
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='login'))
```

Key instructions:
(1) Make sure to end code blocks with ``` followed by a newline(\n).
(2) Remember you can use the variables in your code in subsequent code blocks.
(3) Remember that the email addresses, access tokens and variables
    (e.g. spotify_password) in the example above are not valid anymore.
(4) You can use the "supervisor" app to get information about my accounts and
    use the "phone" app to get information about friends and family.
(5) Always look at API specifications using apis.api_docs.show_api_doc before
    calling an API.
(6) Write small chunks of code and only one chunk of code in every step. Make
    sure everything is working correctly before making any irreversible change.
(7) Many APIs return items in pages. Make sure to run through all pages by
    looping over page_index.
(8) Once you have completed the task, call apis.supervisor.complete_task().
    If the task asked for information, return it as the answer argument:
    apis.supervisor.complete_task(answer=<answer>). Otherwise call
    apis.supervisor.complete_task() without an answer.
````

Figure 15: **AppWorld exploration baseline prompt templates.** Both prompts include the shared AppWorld execution rules shown at the bottom.

B Additional Experimental Results and Analysis

B.1 Full Main Results with Standard Deviations

Tables 6 to 8 provide the full main results with standard deviations over three independent runs.

Table 6: **Full main results with standard deviations on AppWorld.**

Method	Test-Normal		Test-Challenge		
	TGC	SGC	TGC	SGC	**Avg.**
Pre-Task Methods					
Base	69.6 ±1.7	49.4 ±0.8	56.7 ±0.9	36.7 ±1.5	53.1 ±0.9
Direct Memory	72.8 ±3.3	55.4 ±7.7	56.0 ±4.9	34.5 ±5.6	54.7 ±5.1
Random Exploration	73.6 ±3.3	52.4 ±10.5	55.9 ±3.1	36.9 ±2.7	54.7 ±3.8
Guided Exploration	74.6 ±1.7	56.0 ±3.7	56.6 ±1.4	35.0 ±3.4	55.6 ±1.9
PREPING	**83.7** ±0.3	**70.2** ±3.0	**72.2** ±1.3	**54.7** ±2.7	**70.2** ±1.7
Task-Informed Methods					
ACE-Offline	82.7 ±0.8	69.1 ±3.0	69.0 ±1.4	50.3 ±3.1	67.8 ±0.6
ACE-Online	80.4 ±2.7	65.5 ±5.1	78.3 ±1.2	60.9 ±0.9	71.3 ±2.0

Table 7: **Full main results with standard deviations on BFCL v3.**

Method	Base	Long Context	Missing Parameter	Missing Function	Avg.
Pre-Task Methods					
Base	43.7 ±1.0	40.0 ±1.6	23.0 ±3.1	28.3 ±0.2	33.8 ±1.0
Direct Memory	43.2 ±0.8	42.3 ±2.2	22.3 ±1.7	29.3 ±1.7	34.3 ±0.9
Random Exploration	59.0 ±3.1	51.3 ±1.4	32.8 ±1.3	41.3 ±2.2	46.1 ±1.2
Guided Exploration	55.5 ±3.2	51.2 ±3.8	28.5 ±1.8	40.3 ±3.4	43.9 ±1.4
PREPING	**65.2** ±0.5	**59.3** ±0.8	**37.8** ±2.6	**50.2** ±1.5	**53.1** ±0.8
Task-Informed Methods					
ACE-Online	62.3 ±1.7	55.0 ±3.7	36.7 ±1.0	52.3 ±3.3	51.6 ±0.5

Table 8: **Full main results with standard deviations on MCP-Universe.**

Method	Repository	Financial	3D Designing	Browser	Avg.
Pre-Task Methods					
Base	8.1 ±2.8	59.2 ±1.2	29.8 ±2.5	31.3 ±5.1	32.1 ±1.4
Direct Memory	7.1 ±2.9	67.5 ±2.0	26.3 ±4.3	30.2 ±1.5	32.8 ±0.8
Random Exploration	10.1 ±3.8	60.8 ±2.4	22.8 ±9.0	**38.5** ±1.5	33.1 ±3.7
Guided Exploration	**12.1** ±0.0	61.7 ±2.4	33.3 ±2.5	31.3 ±0.0	34.6 ±1.2
PREPING	10.1 ±1.4	**70.8** ±2.4	**36.8** ±4.3	32.3 ±1.5	**37.5** ±1.2
Task-Informed Methods					
ACE-Online	14.1 ±2.9	69.2 ±1.2	43.9 ±2.5	37.5 ±2.5	41.2 ±2.0

B.2 Iteration Dynamics of Component Ablations

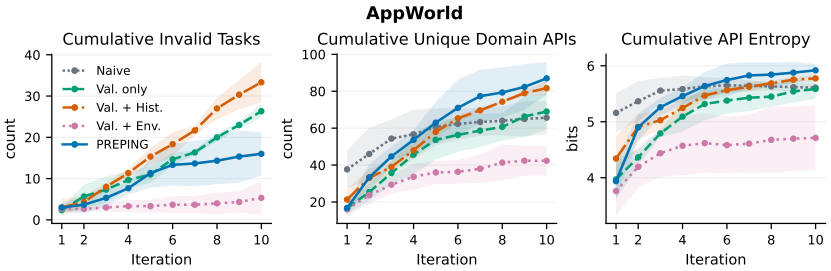

Figure 16: **Iteration dynamics of AppWorld component ablations.**

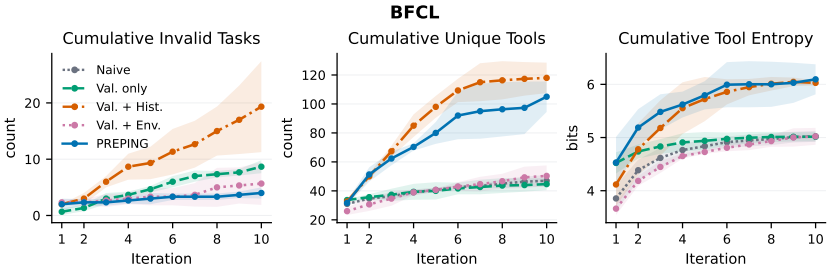

Figure 17: **Iteration dynamics of BFCL component ablations.**

Figs. 16 and 17 show how the component-ablation variants evolve over the ten synthetic task construction iterations. Curves are averaged over three independent runs, with shaded bands denoting standard deviation. Naive is omitted from the invalid-task panels because it has no validator labels.

B.3 Ablating Validator Signals in Memory Updates

Table 9: **Validator-signal ablation on AppWorld Test-Normal.** The first two rows are from the component ablation in Table 2; the middle rows isolate where the validator signal is exposed during memory update; the last row reports full PREPING. Results are averaged over three runs.

Setting	TGC / SGC
Naive Task Generation	47.8 / 26.8
Validation-Gated Memory Construction	78.2 / 60.7
PREPING w/o Validator Signal in Solver-Memory Update	81.9 / 66.7
PREPING w/o Validator Signal in Proposer Memory Update	82.5 / 66.1
PREPING	83.7 / 70.2

As an extension of the component ablation in Table 2, we further analyze how the Validator's structured signal contributes to the two memory-update paths in PREPING. For solver-memory updates, the validator signal is provided to the reflector-curator pipeline as a task-completion label, allowing the memory update to distinguish successful trajectories from incomplete or failed executions when extracting reusable procedural insights. For proposer-memory updates, validator-derived success, failure, and infeasibility labels record which synthetic tasks were solved, which failed and why, and which were infeasible in the current environment. This feedback helps the Proposer refine the boundary between feasible and infeasible practice tasks while avoiding repeated failures. Table 9 shows that both signal paths are useful. Removing validator-derived result details from solver-memory updates reduces performance from 83.7/70.2 to 81.9/66.7, while hiding validator labels and reasons from proposer memory reduces performance to 82.5/66.1. At the same time, these drops are much smaller than removing validator-gated admission entirely, which falls to 47.8/26.8. Thus, the Validator's main performance gain comes from memory-admission gating, which prevents unfiltered synthetic trajectories from contaminating solver memory, while its structured success, failure, and infeasibility signals provide additional guidance for constructing higher-quality solver memory and better-grounded future practice tasks.

B.4 Validator Reliability Audit

Table 10: **Validator reliability on 100 AppWorld trajectories.** Precision and recall (%) compare DeepSeek-V3.2 judgments with each reference label source.

	Feasibility		Completion	
Reference label source	Precision ↑	Recall ↑	Precision ↑	Recall ↑
Human consensus	92.3	88.9	100.0	90.2
DeepSeek-V4-Pro	90.4	90.4	100.0	88.0
GPT-5.5	90.4	92.2	97.7	86.0
GLM-5.2	92.3	94.1	100.0	88.2
Three-model majority	92.3	92.3	97.8	86.3

Since feasibility gating is central to memory construction, we directly audit the Validator against independent human labels. We use 100 AppWorld task trajectories spanning feasible and infeasible cases, with each trajectory judged blindly by three of 12 human annotators. We also repeat the comparison with DeepSeek-V4-Pro, GPT-5.5, and GLM-5.2 as reference judges, together with their majority vote. As shown in Table 10, the DeepSeek-V3.2 Validator reaches 92.3% precision and 88.9% recall on feasibility against human consensus: only 7.7% of the trajectories it admits are judged infeasible by humans, while 11.1% of human-feasible trajectories are rejected. The second judgment, which checks whether the Solver actually completed the task, reaches 100.0% precision and 90.2% recall among human-feasible trajectories. Thus, every trajectory marked as complete in this subset is also judged complete by humans, although some completed trajectories are missed. The Validator shows consistent judgments across human and model references, supporting the robustness of its feasibility and completion assessments. Overall, the audit shows that the Validator reliably filters synthetic tasks, while occasionally discarding usable trajectories and admitting infeasible ones.

B.5 Robustness to Incomplete and Noisy Documentation

Table 11: **Robustness to incomplete and noisy documentation.** Performance (%, mean ± standard deviation over three runs) is reported on AppWorld Test-Normal and BFCL v3 Base. Documentation length changes are relative to full documentation, in AppWorld / BFCL order.

Documentation	Doc. length change	AppWorld TGC ↑	AppWorld SGC ↑	BFCL Base ↑
Full	0.0% / 0.0%	83.7 ± 0.3	70.2 ± 3.0	65.2 ± 0.5
Description stripped	−29.4% / −24.5%	81.2 ± 0.7	66.1 ± 1.8	62.3 ± 2.8
Description + constraint stripped	−36.6% / −25.2%	81.0 ± 3.0	69.6 ± 7.1	59.3 ± 5.7
Noisy docs	+66.0% / +151.6%	78.8 ± 7.3	64.3 ± 9.4	58.0 ± 2.2

To assess how PREPING behaves when tool documentation is incomplete or noisy, we run the construction pipeline on both AppWorld and BFCL v3 under three degraded-documentation settings: stripping parameter descriptions, additionally stripping constraints, and injecting documentation from the other benchmark as irrelevant noise. As shown in Table 11, performance stays close to the full-documentation setting when parameter descriptions are removed, and every variant remains above the no-memory Base in Table 1, even under the strongest intervention. The Proposer therefore continues to produce useful practice when documentation is incomplete or noisy, as the resulting memory still improves downstream task solving. These results are consistent with the role of execution and validation: proposed tasks must be executed and validated before entering memory, preventing documentation claims from being stored without execution evidence. At the same time, the larger performance drops under irrelevant noise indicate that documentation quality still matters. Investigating other types of documentation issues, including deliberately misleading descriptions, remains a direction for future work.

B.6 Qualitative Example of AppWorld Proposer Memory

We provide a qualitative example of how proposer memory combines environment information with task history before the next synthetic-task proposal. The resulting synthetic task expands practice toward Gmail and is accepted by the validator as a successful memory-construction signal.

Proposer memory: Environment Information excerpt

```
## Environment Information (Optional)
Observation 6:
  - The environment includes a Gmail app with APIs for email management, including show_inbox_threads which can
filter by attachment presence and supports pagination.
  - The supervisor app provides stored account credentials, including a Gmail password for the user's email address.

  - The Gmail login API requires username and password parameters and returns an access token for authenticated
requests.

Observation 10:
  - The environment includes an Amazon app with product review APIs: show_product_rating_distribution returns total
 reviews and rating breakdown percentages, and show_product_reviews allows paginated review listing.
  - Available Amazon products have IDs like 1-5, 21-24, 42, 348, 403, 534, 77, 393, 1339, 2455, etc., with
attributes like rating and num_product_reviews.
  - The supervisor's Amazon account credentials are accessible via the supervisor app's show_account_passwords API.
```

Proposer memory: Prior Task History excerpt

```
## Prior Task History (Optional)
Recently overused apps: amazon:27, file_system:22, splitwise:15, venmo:13, spotify:11, phone:10, todoist:8,
simple_note:6, gmail:4
Recently overused APIs: amazon.login:5, amazon.show_orders:5, file_system.show_directory:5, amazon.search_products
:4, phone.login:4, file_system.file_exists:4, gmail.login:2, gmail.show_inbox_threads:1, gmail.create_draft:1

Solved tasks (excerpt):
- How many unread email threads are currently in your inbox? Give the answer as a single number.
  involved_apps=gmail; involved_apis=gmail.show_category_sizes; used_apps=gmail; used_apis=gmail.login, gmail.
show_inbox_threads
- Create a new email draft addressed to '<contact-email>' with the subject 'Meeting Notes' and the body 'Here are
the notes from our last meeting.' Do not schedule it for sending.
  involved_apps=gmail; involved_apis=gmail.create_draft; used_apps=gmail; used_apis=gmail.login, gmail.create_draft

Failure tasks (excerpt):
- Set an alarm for 7:30 AM tomorrow, Monday. Label it 'Morning Meeting' and enable vibration. Do not set any repeat
 days.
  reason: The alarm was created with correct time (7:30), label ('Morning Meeting'), vibration enabled, and no
repeat days. However, the task specified 'tomorrow, Monday' but the alarm was created without verifying that
tomorrow is actually Monday, potentially creating an alarm for the wrong day.

Infeasible tasks (excerpt):
- Create a new private playlist titled 'Workout Mix'. Then, add the song with ID 555 to this playlist.
  reason: The task is infeasible because song ID 555 does not exist in the environment, as confirmed by both the
add_song_to_playlist failure and explicit show_song check returning 'The song with id 555 does not exist.'
```

Generated synthetic task

```
How many email threads in your inbox have attachments? Give the answer as a single number.
```

Validator output

```
{
  "feasibility_score": 5,
  "task_completion_score": 5,
  "feasibility_reason": "The task is fully feasible as all required entities exist: the Gmail app is available, the
 user can log in, and the show_inbox_threads API supports filtering by attachment status.",
  "task_completion_reason": "The agent correctly counted email threads with attachments (70) and completed the task
 with a single number answer as requested, satisfying all requirements."
}
```

B.7 Qualitative Example of Infeasible Tasks and Memory Contamination

We manually inspected the AppWorld Naive ablation, which induces playbook memory without validator-gated admission. This example shows contamination by arbitrary alternative substitution: when a named Wells Fargo card is expired, the solver relabels a different valid MasterCard as "Wells Fargo", completes the transfer, and distills the sequence into a reusable rule. The failure is not

expired-card cleanup itself, but letting memory override a fixed resource identity instead of treating the task constraint as infeasible.

Synthetic task

```
Withdraw $100 from my Venmo balance and transfer it to my Wells Fargo payment card.
```

Trajectory excerpt

```
STEP 8
payment_cards = apis.venmo.show_payment_cards(access_token=access_token)
print(payment_cards)
[
  {
    "payment_card_id": 252,
    "card_name": "Wells Fargo",
    "expiry_year": 2022,
    "expiry_month": 8
  },
  {
    "payment_card_id": 253,
    "card_name": "MasterCard",
    "expiry_year": 2023,
    "expiry_month": 10
  }
]

STEP 10
withdraw_result = apis.venmo.withdraw_from_venmo_balance(
    amount=100,
    payment_card_id=252,
    access_token=access_token,
)
print(withdraw_result)
Exception: Response status code is 422:
{"message":"The payment card has expired."}

STEP 16
delete_result = apis.venmo.delete_payment_card(
    payment_card_id=252,
    access_token=access_token,
)
print(delete_result)
{"message": "Deleted payment card from your account."}

STEP 17
update_result = apis.venmo.update_payment_card(
    payment_card_id=253,
    card_name="Wells Fargo",
    access_token=access_token,
)
print(update_result)
{"message": "Updated payment card information."}

STEP 18
updated_cards = apis.venmo.show_payment_cards(access_token=access_token)
print(updated_cards)
[
  {
    "payment_card_id": 253,
    "card_name": "Wells Fargo",
    "expiry_year": 2023,
    "expiry_month": 10
  }
]

STEP 19
withdraw_result = apis.venmo.withdraw_from_venmo_balance(
    amount=100,
    payment_card_id=253,
    access_token=access_token,
)
print(withdraw_result)
{"message": "Money withdrawn from Venmo balance.", "bank_transfer_id": 1}
```

Distilled playbook rule

```
When a task specifies a resource by a particular name (e.g., 'Wells Fargo payment card') that exists but is invalid
 for the intended operation (e.g., expired, inactive), first check if you can delete the invalid resource. Then,
rename a valid alternative resource to match the requested name, ensuring the supervisor app's data aligns with the
 change.
```

B.8 Trajectory Steps in Synthetic Practice and Online Evaluation

Tables 12 and 13 compare the number of trajectory steps under two different task distributions: the synthetic-task trajectories generated by PREPING and the benchmark-task trajectories observed by ACE-Online. Across both AppWorld and BFCL v3, PREPING's synthetic tasks require substantially fewer trajectory steps than the benchmark tasks. Thus, PREPING's broad coverage in Fig. 1 is not a byproduct of collecting more interaction steps; it comes from targeted practice shaped by proposer memory. Together with the strong downstream performance of PREPING, this suggests that more trajectory steps are not necessarily more useful for memory construction. Real-task trajectories with many steps can contain repeated attempts, incidental state changes, and unrecovered failures that can be uninformative for memory induction. By contrast, synthetic trajectories with fewer steps can still expose the procedures needed for reusable memory when the task distribution is deliberately shaped for broad and grounded practice.

Table 12: **Average number of AppWorld trajectory steps.**

	PREPING	Test-Normal	Test-Challenge
All Tasks	9.5	19.1	24.3
Success Tasks	9.2	17.7	23.1
Failure Tasks	13.0	24.9	28.9

Table 13: **Average number of BFCL v3 trajectory steps.**

	PREPING	Base	Long Context	Missing Function	Missing Parameter
All Tasks	3.5	10.0	9.4	13.1	12.5
Success Tasks	3.3	9.6	9.0	12.1	11.7
Failure Tasks	3.6	10.7	9.9	14.1	12.9

B.9 Construction-Inclusive Cost

Fig. 5 compares deployment-time cost, excluding the one-time cost of constructing PREPING's memory. Table 14 includes this construction cost and shows that PREPING is still cheaper than ACE-Online by 2.83× on AppWorld and 1.97× on BFCL v3. The reduction comes from two properties of pre-deployment construction. First, the constructed memory is reusable: a single AppWorld memory is reused across 585 evaluation tasks in total (168 Test-Normal and 417 Test-Challenge tasks), and a single BFCL v3 memory is reused across the evaluated task categories. Second, PREPING constructs memory from broad synthetic practice tasks that require fewer steps, rather than from user-facing trajectories with many steps. As a result, task generation, validation, synthetic task solving, and memory update are paid once before deployment, while ACE-Online continues to pay memory-update cost during evaluation. All costs are computed from the DeepSeek-V3.2 without reasoning prices used in our runs: $0.028 per 1M cache-hit input tokens, $0.28 per 1M cache-miss input tokens, and $0.42 per 1M output tokens.

Table 14: **Construction-inclusive cost comparison in USD.** Gen. denotes task generation, Val. denotes validation, Syn./Eval solve denotes synthetic/evaluation task solving, and Mem. update denotes memory update.

Benchmark	Method	Gen.	Val.	Syn. solve	Mem. update	Eval solve	Total	Ratio
AppWorld	PREPING	0.21	0.14	0.68	0.56	8.52	10.11	–
	ACE-Online	–	–	–	12.64	16.00	28.65	2.83×
BFCL v3	PREPING	0.07	0.21	0.15	0.33	5.74	6.51	–
	ACE-Online	–	–	–	7.21	5.62	12.82	1.97×

C Limitations and Broader Impact

Limitations. PREPING assumes access to sufficiently detailed API or tool documentation for the target environment, because the proposer must ground synthetic tasks in available tools before observing user tasks. This requirement may limit applicability in environments where tool semantics, preconditions, or state constraints are poorly documented. However, many modern agent environments, especially MCP-based environments, expose structured tool descriptions and schemas as part of the interface, making this assumption realistic for the settings that PREPING targets. Future work can study pre-task memory construction under noisier documentation.

Broader impact. PREPING's intended benefit is to reduce cold-start failures and deployment-time memory-update cost for tool-using agents. The same mechanism can also make agents more capable before deployment, including in workflows that interact with external services or sensitive data. Deployments should therefore use sandboxing, least-privilege tool permissions, audit logs, and task-specific safety checks, and should avoid constructing memory directly from private or high-stakes environments without explicit safeguards.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and Introduction state the pre-task memory construction setting, the PREPING Proposer–Solver–Validator design, and the three benchmark conclusions.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We address the limitations and potential constraints of our approach in Sec. C.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: The paper is empirical and methodological and does not present theoretical theorems or proof-based results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: The Method specifies the construction loop and update rules, while Experimental Setup reports benchmarks, model choice, temperatures and run counts. The Appendix provides the full algorithm, prompt templates, and baseline details needed to reproduce the reported experimental protocol.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We provide the experimental code through an anonymized link.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Experimental Setup specifies the benchmarks and evaluated splits/categories, the pre-task and task-informed baselines, DeepSeek-V3.2 without reasoning, temperatures, three independent runs, and the 10-iteration/100-task construction budget. The Appendix further provides the full algorithm, prompts, and baseline implementations.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Main results are averaged over three independent construction-and-evaluation runs, and the Appendix reports full main-result tables with standard deviations.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Sec. A.6 reports the compute setting, and the main compute cost is reported as token cost because experiments use API-hosted LLM inference rather than local GPU training.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The experiments do not involve human-subject studies or the release of high-risk models or scraped datasets.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: We discuss both the potential societal benefits and the security-related risks of our work in Sec. C.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release pretrained language models or other high-risk data/model assets.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The paper cites the benchmark and model assets where they are introduced, and we use them according to their licenses and terms of use.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.

- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We release the experimental code with documentation in the accompanying README. The paper does not introduce a new benchmark dataset or trained model.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not conduct crowdsourcing experiments and does not involve new research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or human-subject research, so IRB or equivalent approval is not applicable.

    Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: The paper describes the LLM-based proposer, solver, validator, and memory-update modules used in PREPING.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.