ENABLING TIMELY GUIDANCE BEFORE SKILL RETRIEVAL: RETAINING HELPFUL WARM TIPS IN AGENT CONTEXT

Anonymous authors
Paper under double-blind review

ABSTRACT

Reusable skills help LLM-based agents solve complex tasks, but the agent must receive guidance before it commits to an ineffective approach. Existing skill mechanisms often expose only metadata and load full content on demand, leaving useful guidance unavailable until the agent decides to retrieve it. General memory methods can incur substantial maintenance overhead, while keeping guidance in conversation context risks repeatedly exposing the agent to irrelevant or harmful advice. We propose TipsWarm, a mechanism that complements existing skill mechanisms by maintaining a budgeted pool of skill-derived keypoints, or *warm tips*, for selective injection into the context of every message turn. By separating event-triggered LLM assessment from inexpensive per-turn screening, it makes transferable skill guidance readily available while controlling maintenance costs. In three coding and iterative task-execution benchmarks, TipsWarm achieves the highest task success rate while remaining time-efficient, compared to recent skill and general memory baselines. [1]

1 INTRODUCTION

Memory of experience is essential for LLM-based agents to accomplish complex tasks, such as coding (Yang et al., 2024) and interactive decision making (Yao et al., 2023; Shinn et al., 2023). A skill structures reusable experience into actionable guidance, helping agents follow established procedures or avoid known pitfalls (Li et al., 2026). Skills drafted manually in advance may miss keypoints or even provide misleading guidance for new tasks. Considerable effort has been devoted to dynamically generating skills from agent experience, such as interaction trajectories (Zhao et al., 2024; Yang et al., 2026; Ni et al., 2026), with a goal of producing guidance that transfers to similar tasks. As agents encounter new tasks, however, previously generated skills may prove flawed or become outdated. Recently, several studies have proposed methods for evolving skills throughout the task-solving process. For example, Memento-Skills (Zhou et al., 2026) updates and expands its skill library from new experience, and SkillRL (Xia et al., 2026) recursively co-evolves a hierarchical skill bank with the agent policy. These updates help skills adapt to new experience and improve their potential to transfer to future similar tasks.

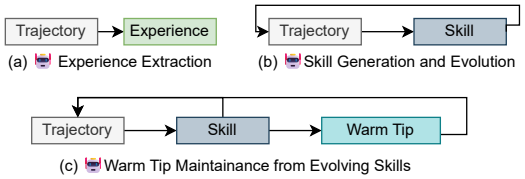


Figure 1: Thumbnail comparison of different agent experience paradigms.

A challenge of using skills is that useful experience can be buried in a large skill repository and may not be retrieved in time to guide the agent's decisions. This problem arises when the conversation context initially contains only skill metadata, such as names and descriptions, and the agent retrieves full skill content only when it deems retrieval necessary. Before deciding to retrieve a relevant skill, the agent may waste effort on an ineffective approach, reducing its chance of solving the task within available budgets of message turns. Our hypothesis is that *keeping relevant skill keypoints readily available before full-skill retrieval helps agents make productive decisions earlier*. This

[1] The source code and results are available in the attachment for blind review.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

early guidance can help the agent avoid wasted effort and solve tasks more effectively. For an analogy, imagine a programmer returning to experiment scripts written long ago, with the option specifications buried in a stack of documents. The programmer may struggle to choose the right options even though the documentation is available for reference. The programmer may choose the wrong settings before realizing which documentation to consult. A short note beside the scripts, listing key constraints and common mistakes from previous experiments, can guide those choices immediately, while the full documentation remains available when needed. We describe helpful skill keypoints, such as the short note above, kept readily accessible in the conversation context as *warm tips*, with a distinction from the *cold skills* that must be dug up from memory when needed. Retaining useful warm tips in the context can help guide the agent's initial decisions before it recognizes the need to dig through the repository for a full skill. Fig. 1 compares different paradigms of experience memory in a thumbnail.

Existing agent memory mechanisms also manage reusable experience in the conversation context. One general approach is to retrieve query-relevant information from memory and inject it into the context, independently of the skill mechanism. For instance, A-MEM (Xu et al., 2025) dynamically links and updates memory notes and retrieves query-relevant ones. Dynamic Cheatsheet (Suzgun et al., 2026) organizes reusable strategies and code snippets into a compact memory that conditions subsequent queries. Managing experience separately from skills raises concerns about both effectiveness and efficiency. On the one hand, a separate memory mechanism may overlook useful guidance already captured in skills, including lessons and warnings that could prevent ineffective actions. On the other hand, maintaining a growing memory store and retrieving relevant information usually incur substantial overhead. For example, according to our observations in preliminary experiments, by updating, embedding, and retrieving information in almost every turn, A-MEM required twice as many LLM calls and took approximately 14-25$\times$ as much wall-clock time as the dynamic skill evolution baseline. Drawing these tips from existing skills offers a way to reuse the guidance in existing skills with controlled additional overhead.

However, maintaining warm tips in context requires attention to both their quality and cost. Warm tips require more careful selection than stored skills because they can directly influence the agent's planning and actions in every turn. Maintaining and injecting tips must also remain affordable. While remaining compatible with the existing skill mechanism, warm tips should remain available in every turn for immediate guidance without requiring costly selection or updates each time.

We propose **TipsWarm**, an in-context warm-tip management mechanism with careful designs for effectiveness and efficiency. TipsWarm continuously maintains a pool of candidate warm tips under a size budget and manages their quality through three operations. First, when a skill is accessed, generated, or updated, the *filtering* operation extracts keypoints and admits those judged by an LLM to be potentially helpful for future similar tasks. Then, in every user message turn, the *injection* operation selects warm tips from the pool based on their relevance and past usefulness and adds them to the context. Because injected tips can influence subsequent actions, TipsWarm uses scope matching and historical utility estimates to reduce exposure to potentially misleading guidance. Finally, after task execution, the *attribution* operation uses an LLM to assess how injected tips helped or misled the agent, based on its actions and task outcomes. These assessments guide future tip selection during injection. TipsWarm limits overhead by coordinating the timing and cost of these operations. Filtering and attribution invoke an LLM only at skill events and task completion, respectively, while per-turn injection reuses their output through inexpensive heuristic rules. TipsWarm complements the existing skill mechanism and integrates into the agent loop downstream of skill generation and evolution. Across three coding and interactive task-execution benchmarks, TipsWarm improves final task success over the existing skill mechanism. Compared to recent general memory methods, TipsWarm improves task success with better success-time trade-offs. Major contributions of this work are summarized as follows:

- We design TipsWarm to keep useful skill keypoints available in context, enabling them to guide agents before full skills are retrieved. It complements existing skill mechanisms by reusing their guidance.

- TipsWarm coordinates the filtering, injection, and assessment of warm tips for both effectiveness and efficiency. It uses LLM judgments only at skill events and task completion, and fast screening in every turn to limit agents' exposure to harmful tips.

- We show that selectively exposing existing skill guidance improves final task success across three benchmarks, and that TipsWarm enables agents with weaker LLMs to approach the performance of those with stronger ones and augments skills learned without warm tips.

2 Related Work

Agent experience. LLM agents commonly interleave reasoning with environment interaction (Yao et al., 2023). Early approaches reuse experience at different levels of abstraction. Synapse stores and retrieves complete trajectories as in-context exemplars (Zheng et al., 2024), whereas Reflexion converts task feedback into verbal reflections that are retained across trials (Shinn et al., 2023). Later work further abstracts experience into reusable procedural knowledge. For example, ExpeL (Zhao et al., 2024) distills cross-task insights from successes and failures. Voyager (Wang et al., 2023) accumulates an embedding-indexed library of executable programs for lifelong embodied exploration. Agent Workflow Memory (Wang et al., 2025) induces recurring action routines for web agents. These methods established that compact procedures can transfer more effectively than replaying raw trajectories. They nevertheless generally retrieve experience or workflows in response to the current task. Our work studies a complementary failure mode where an agent may plan incorrectly before it requests or successfully retrieves the relevant procedure. We therefore distill only decision-critical points from persistent skills and keep a budgeted subset continuously available in the working context.

Skill generation and evolution. Recent systems treat natural-language instructions, code, and supporting artifacts as explicit skills that can be created and revised without updating the base model. One line of work focuses on skill synthesis. Agent traces are distilled into reusable skills (Alzubi et al., 2026), consolidated across diverse traces (Ni et al., 2026), or refined through verification (Zhang et al., 2026). A second line studies continual skill evolution. These methods update skill libraries as new interactions arrive, either for an individual agent (Yang et al., 2026; Zhou et al., 2026), across users (Ma et al., 2026), or across complementary forms of experience (Jiang et al., 2026). A third line couples skills with model learning. It uses evolving skills to guide policy optimization (Xia et al., 2026) or distills skill guidance into model parameters (Wang et al., 2026). The above lines improve how skills are created, maintained, and learned, whereas our work addresses a later stage in the skill lifecycle. TipsWarm is agnostic to the upstream skill generator and operates after skill creation by managing a small set of skill-derived warm tips that remains visible across turns. Rather than competing with their learning objectives, it can consequently augment skill libraries, whether they are static, automatically generated, or continually evolving.

Long-term memory and context management. One line of work equips agents with long-term memory and retrieves relevant records when needed. Retrieval-augmented generation (RAG) (Lewis et al., 2020) provides the canonical pattern of selecting records from non-parametric storage and concatenating them with the input. Agent memory systems extend this pattern to episodic feedback (Shinn et al., 2023), learned action values (Zhang et al., 2023), and reusable workflows (Wang et al., 2025). A-MEM (Xu et al., 2025) further organizes interactions as linked, evolving memory notes and retrieves relevant notes for each query. Although retrieval makes large memories scalable, its benefit depends on whether retrieval is initiated and whether the correct item ranks highly. A second line of work manages the information already placed in the context window. Models can underuse relevant evidence when it is poorly positioned in a long context (Liu et al., 2024), which motivates prompt-compression methods that preserve salient information under a token budget (Jiang et al., 2023). Most closely related to our setting, Dynamic Cheatsheet (Suzgun et al., 2026) continually organizes strategies, solution patterns, and code snippets into a compact memory that conditions subsequent queries.

TipsWarm is a variant of both lines of work and is specially designed for skills. In contrast to A-MEM, which retrieves query-relevant interaction memories, TipsWarm proactively maintains skill-derived techniques in the active context without waiting for query-time retrieval. In contrast to Dynamic Cheatsheet, which learns a general memory directly from model inputs and outputs, TipsWarm extracts decision-critical points from explicit skill artifacts and schedules them within a fixed budget across agent conversation sessions. Its distinctive design is to inject this working set at every user turn so that critical techniques can influence planning before an agent decides to retrieve a full

skill. Thus, persistent skills remain the comprehensive source of knowledge, while warm tips form the readily accessible layer between long-term memory and active reasoning.

3 Background and Problem Definition

Background. This study focuses on loop-based agent conversation (Yao et al., 2025), which is the pervasive paradigm of general agent workflows as compared to the graph-based alternative (Zhang et al., 2025). An agent solves a task through a conversation session consisting of multiple turns. Here, a turn refers to one cycle in which the agent processes a user message with the available context and produces a response. We use "loop" and "turn" interchangeably. A typical conversation loop is depicted in Fig. 2 (a). The response may contain reasoning, a plan, or a tool-call request, including loading, generating or updating skills via skill tools. The history of messages, agent responses, and tool results forms most of the conversation trajectory, which then becomes part of the context for the next turn.

We next describe the context provided to the agent. The conversation context includes both session-fixed and turn-specific information. Session-fixed information typically includes system and user profile prompts. Skill metadata and other memory content may also remain fixed if the agent does not refresh them during the session. Turn-specific information is updated as the conversation progresses. It mainly includes the conversation trajectory, which may contain skill content generated or loaded in previous turns. In this study, we introduce warm tips as a new component of the turn-specific context. Warm tips are keypoints extracted from skills and maintained in the context across turns.

Problem Definition. We formally define warm tips within an agent conversation session. In turn t, we represent the context relevant to the skill and warm-tip mechanisms as a tuple $(\mathcal{P}, \mathcal{M}_t, \mathcal{T}_t, \mathcal{W}_t)$, where $\mathcal{P}$ contains the system and user profile prompts, $\mathcal{M}_t$ contains the skill metadata, $\mathcal{T}_t$ is the conversation trajectory before the current turn, and $\mathcal{W}_t$ is the set of warm tips injected in this turn. Let a skill be $s = (m, \mathbb{C})$, with metadata m and content represented as a set of skill points $\mathbb{C} = \{c_1, c_2, \ldots, c_k\}$. Each skill point c_i is a unit of guidance, such as an instruction, a constraint, or a warning about a pitfall. The agent's skill repository at turn t is denoted by $\mathbb{S}_t = \{s_1, s_2, \ldots, s_n\}$. Both the number of skills and their contents may change over time. The repository's skill metadata are $\mathcal{M}_t = \{m_i \mid (m_i, \mathbb{C}_i) \in \mathbb{S}_t\}$. Skill metadata are typically loaded when the session starts and remain unchanged unless refreshed, while skill content can be generated or updated during the session.

The conversation trajectory $\mathcal{T}_t$ can include content from previously loaded skills. Let $\mathcal{S}_t \subseteq \bigcup_{\tau<t} \mathbb{S}_\tau$ denote the skill versions loaded before turn t. These versions were retrieved based on the contexts and user messages available in earlier turns. Let $\mathcal{U}_t$ be the user message at turn t, and let f_{load} represent the cumulative skill retrieval process. We summarize the process of loading skills below, where a subscript $< t$ denotes the corresponding history before turn t:

$$\mathcal{S}_t = f_{load}(\mathbb{S}_{<t}; \mathcal{P}, \mathcal{M}_{<t}, \mathcal{T}_{<t}, \mathcal{W}_{<t}, \mathcal{U}_{<t}). \tag{1}$$

In contrast, warm tips are selected skill points injected directly into the current context. We denote them by $\mathcal{W}_t \subseteq \bigcup_{(m,\mathbb{C})\in\mathbb{S}_t} \mathbb{C}$, so each tip comes from a skill in the current repository. Let f_{warm} denote the warm-tip maintenance process. It determines the tips for turn t from the current skills, context, user message, and previously injected tips:

$$\mathcal{W}_t = f_{warm}(\mathbb{S}_t; \mathcal{P}, \mathcal{M}_t, \mathcal{T}_t, \mathcal{W}_{t-1}, \mathcal{U}_t). \tag{2}$$

Skills, warm tips, and memory are coherent but distinct in representation and meaning. Loaded skills $\mathcal{S}_t$ provide full skill content, which may include complete procedures for solving tasks. Warm tips $\mathcal{W}_t$ provide selected points from skills, such as useful techniques or warnings about pitfalls. The skill points that exist in the loaded skills and warm tips may or may not overlap, depending on the skill content and the warm-tip selection process. General memory mechanisms may also supply information to the context. The warm-tip mechanism specifically manages skill-derived guidance and can coexist with these mechanisms.

Given the context and user message, the LLM produces the agent response $\mathcal{O}_t$:

$$\mathcal{O}_t = f_{LLM}(\mathcal{P}, \mathcal{M}_t, \mathcal{T}_t, \mathcal{W}_t, \mathcal{U}_t). \tag{3}$$

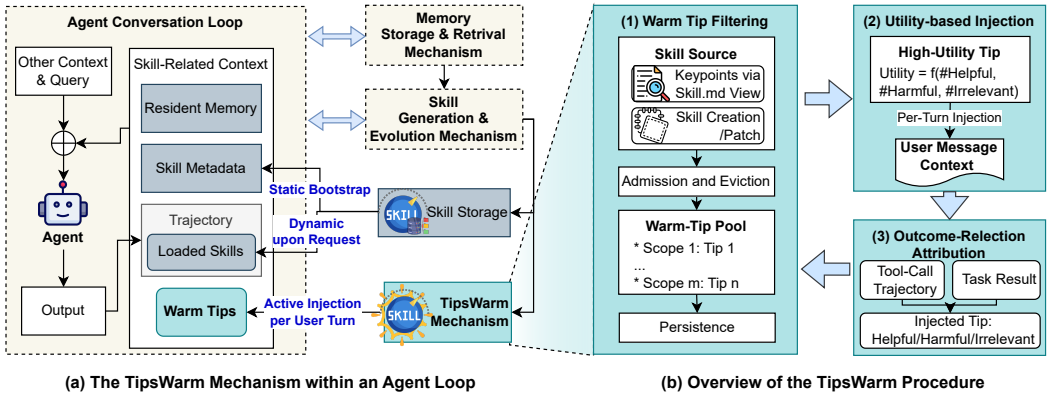


Figure 2: Overview of a) TipsWarm in an agent conversation loop and b) TipsWarm's operations

The objective of this study is to design a warm-tip mechanism f_{warm} that helps the agent solve new, similar tasks more efficiently and accurately. It operates downstream of skill generation, updates, and retrieval, and can be used alongside general memory mechanisms. The mechanism must select useful skill points while limiting the cost of maintaining and injecting them.

4 METHOD

TipsWarm consists of three major operations as shown in Fig. 2 (b). 1) The *filtering* operation maintains a budget-limited pool of potentially helpful warm tips with admission and eviction strategies for skill keypoints every time a skill is viewed, generated or updated. 2) The *injection* operation quickly screens tips for relevance and utility, then adds the selected tips to the conversation context in every user message turn. 3) The *attribution* operation assesses the contribution of injected tips using task outcomes and updates their utility records for future selection. These operations connect skill-derived guidance, its use during a task, and feedback from that task.

Design principles. TipsWarm must keep useful guidance available across turns (effectiveness) while limiting the cost of maintaining it (efficiency). Refining skill quality can require substantial additional work. For example, a typical approach is to iteratively generate and evaluate candidate skills, retaining those that yield better task performance (Alzubi et al., 2026; Zhang et al., 2026). 1) In contrast, the first design principle of TipsWarm is to selectively reuse existing skills, which provides a source of guidance without a separate loop for generating and evaluating skill variants. TipsWarm therefore focuses its additional effort on selecting guidance for immediate exposure and updating its estimated usefulness from task feedback. 2) A second design principle is to separate costly tip assessment from frequent tip injection for further runtime efficiency. Filtering uses LLM judgment when skills are viewed, generated, or updated, while attribution uses it after task completion, both of which are triggered by relatively infrequent events. Injection runs in every turn using fast heuristic rules and the accumulated utility records, without an additional LLM judgment. This allows immediate guidance to remain available throughout the conversation without repeating costly assessments in every turn. Below, we present the detailed design of TipsWarm for high effectiveness and efficiency.

4.1 WARM TIP FILTERING

Warm tips in the conversation context should be kept concise and instructive. Keeping full skill content in every turn can consume substantial context space and expose the agent to information that is irrelevant or even harmful for new tasks. To limit context use and reduce misleading guidance, we maintain a budget-limited pool of warm tips extracted from skills. Because injected guidance can influence successive decisions, a misleading tip may cause repeated wasted effort. Therefore, we design this operation to guard against potentially harmful guidance while admitting tips likely to help on future similar tasks.

Filtering maintains the warm-tip pool by combining efficient candidate extraction with LLM-based assessment of transferability. First, it uses regular expressions (regex) to match skill sections that are heuristically considered important. Examples include sections on pitfalls, best practices, and limitations, as well as explicitly tagged sections when the skill generator provides such tags. Regex matching efficiently extracts candidate keypoints, but does not determine whether their advice is helpful or harmful. Leaving out some helpful tips is less of a concern here, as the regular expression set can be extended to cover additional potentially helpful tips. Then, an LLM judges whether candidate tips are likely to help on unseen tasks of a similar kind. The admission strategy adds promising tips to the pool, while the eviction strategy removes less useful or potentially harmful tips when the pool reaches its budget. Warm tips are tagged with scope keywords for later injection screening. The admission and eviction prompts are provided in Appendix A. The warm-tip pool is stored persistently for reuse across conversation sessions. Conceptually, the filtering procedure can be represented as follows:

$$\text{Skills} \xrightarrow{\text{regex}} \text{Candidate Tips} \xrightarrow{\text{LLM judge with budget}} \text{Warm-Tip Pool}. \tag{4}$$

Filtering limits additional work by following the agent's existing skill operations. It runs only when the agent views, modifies, or generates a skill, so turns without these events require no filtering. The pool budget limits the number of available tips, and injection screening to be introduced soon further restricts the subset added to the context. We cap the total number of tips in the pool (12 in our experiments) and the number contributed by each skill. These limits control the amount of guidance retained and prevent a single skill from dominating the pool.

4.2 Utility-aware Injection

The injection operation selectively exposes the pool to the conversation context in every turn for immediate guidance. The above admission of tips to the pool does not guarantee usefulness for the current task. Injection therefore applies a second screening based on task relevance and historical utility to determine which tips in the pool enter the context. Injected tips add tokens to each turn, so we constrain this overhead via the pool budget and injection quota.

Because injection runs in every turn, relevance and utility are evaluated using inexpensive heuristics. We first match the scope keywords assigned during filtering against the task query to exclude tips from unrelated scopes. For each scope-relevant tip, we use counts of past tasks in which it was labeled helpful, harmful, or irrelevant, which are updated by the attribution operation as described in the next subsection. We derive the utility score of each tip simply by positively weighting the helpful counts and negatively weighting the harmful and irrelevant counts. We then select scope-relevant tips with high utility scores up to the injection quota and insert them immediately before the user message. This reuses past assessments to guide current selection without asking an LLM to reassess the pool in every turn. Conceptually, the procedure for screening warm tips to be injected into the context can be represented as follows:

$$\text{Warm-Tip Pool} \xrightarrow{\text{scope match}} \text{Relevant Tips} \xrightarrow{\text{utility screening}} \text{Injected Tips}. \tag{5}$$

4.3 Outcome-Reflection Attribution

Useful utility scores require feedback on whether a tip helped or hindered task execution. When several tips are injected together, the task outcome alone cannot distinguish their individual contributions. A successful task does not imply that every tip helped, and a failed task does not imply that every tip was harmful.

Therefore, TipsWarm combines task outcomes with evidence from the agent's actions in an LLM-based assessment. The judge considers success and, when available, efficiency and reliability measures alongside tool-call events that indicate whether the agent benefits from a tip. Reprocessing the complete conversation trajectory for attribution can add non-negligible token cost. In comparison, a compact record of tool-call events retains key evidence of the agent's actions while omitting lengthy observations and raw tool outputs. The judge uses this evidence to assess whether a tip plausibly helped or caused wasted effort. In the attribution operation, each injected tip is counted as helpful, harmful, or irrelevant depending on LLM judgment, whose prompt is provided in Appendix A, and

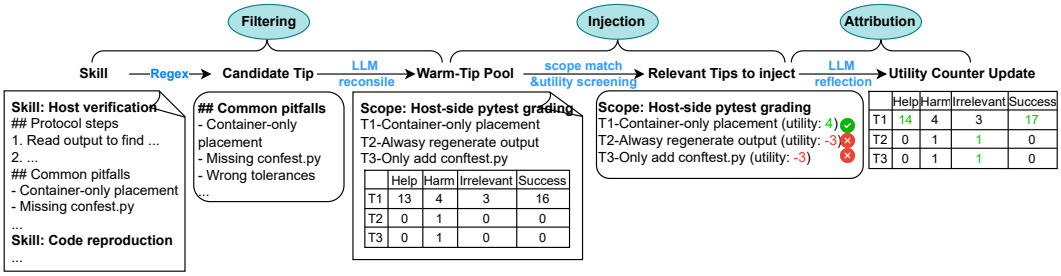


Figure 3: An illustrative example of how warm tips are processed and represented across operations.

the other tips in the pool are counted as irrelevant. Conceptually, the attribution procedure is:

$$(\text{Injected Tips, Tool-Call Events, Task Outcome}) \xrightarrow{\text{LLM judge}} [\text{Helpful}|\text{Harmful}|\text{Irrelevant}]. \quad (6)$$

$$\text{Non-injected Pool Tips} \longrightarrow \text{Irrelevant}. \quad (7)$$

For a concrete understanding of the above procedure, we illustrate the representation forms of warm tips across the operations in Fig. 3, using an example run on SkillsBench (Li et al., 2026).

5 Evaluation

5.1 Experiment Settings

Benchmarks and Splits. We use three multi-turn problem-solving benchmarks. As evaluations should operate in the memory-equipped setting, all benchmarks use a warmup task split for initial skill generation and evolution and a test split of similar tasks for continuous memory evolution and performance evaluation. **1) SkillsBench** (Li et al., 2026) is a comprehensive benchmark to evaluate agents' ability to leverage skills to solve coding tasks to pass test cases. It contains 88 tasks across eight professional domains, solving which requires coding, debugging, and testing in development environments. We randomly split the tasks in a 7:3 ratio stratified by domain, resulting in 65 warmup tasks and 23 test tasks. **2) AlfWorld(-Text)** (Shridhar et al., 2021) requires an agent to interact with a text-based game environment by taking recurring actions in space to complete household routines. Considering the homogeneity in task actions and environment, we use only the first 50 tasks out of the total 3,553 tasks in AlfWorld's train set as warmup tasks, but all 134 tasks in its unseen set as test tasks for evaluation. This setting can highlight the agent's ability to transfer skills distilled from a smaller set of warmup tasks to a large set of test tasks. **3) AppWorld** (Trivedi et al., 2024) requires an agent to complete mobile app tasks by programming with various provided APIs. It combines characteristics of the above two benchmarks, which involve coding and debugging to pass tests, as in SkillsBench and exploration through interaction with an external environment, as in AlfWorld. All 90 tasks in the released train set are used as warmup tasks and all 168 tasks in the test-normal set are used as test tasks.

Metrics. **1) Accuracy:** We use task success rate within a turn budget k ($Success@k$) to measure the accuracy at individual turn budgets, accompanied by the Area under the Curve (AUC) of success rate from turn 1 to the maximum turn budget to reflect the cumulative performance. **2) Time Cost Efficiency:** We use the pair consisting of task success rate and task completion time $(Success@k, Time)$ as a measure. Completion time is averaged over successful tasks only. We also include results of token cost efficiency in Appendix C, with no obviously consistent advantage across methods.

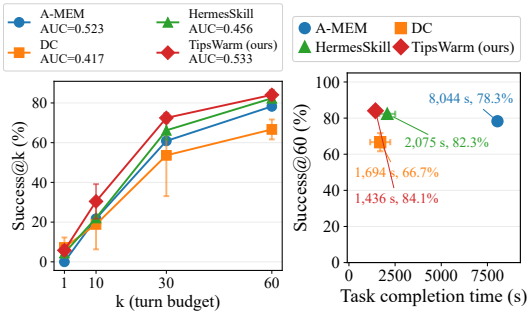
Baselines. We use the following baselines for skill or memory management, which are all integrated within the Hermes agent system (her, 2026), a ReAct-style (Yao et al., 2023) tool-using agent with an initial skill bundle and periodic skill generation and update. **1) HermesSkill** represents Hermes' default skill mechanism that dynamically loads skills into the context upon explicit skill visiting operations. HermesSkill matches a direct comparison with TipsWarm, which retains its

skill and memory mechanisms adding warm-tip management across turns. Besides this, we compare with **2) A-MEM** (Xu et al., 2025) and **3) Dynamic Cheatsheet (DC)** (Suzgun et al., 2026), which represent SOTA general memory baselines for long-term memory retrieval and context management, respectively. They upgrade the agent's default memory mechanism and can be considered as alternative counterparts to TipsWarm.

Running Details. We deploy Qwen3.6-27B (qwe, 2026) as the agent's backbone LLM, with reasoning and tool-call abilities enabled and a maximum context length of 131,072. A detailed list of major hyperparameters for TipsWarm is provided in Appendix B.1. For each experiment, after running once on the warmup tasks, the agent checkpoints its environment and proceeds to the test tasks. Except for A-MEM, which densely updates its experience and takes significantly longer than other methods, all test experiments are run three times to report the average performance with standard deviation. Considering the trajectory length required for each benchmark, the maximum allowed turns are 60 for SkillsBench, 50 for AlfWorld, and 40 for AppWorld. All experiments are conducted on a server with a single NVIDIA A800 GPU with 80 GB of memory and 128-core CPUs of Intel Xeon Platinum 8468. The running costs of different methods on different benchmarks are listed in Appendix B.2. TipsWarm uses a comparable amount of tokens but much less wall-clock time for warmup than the general memory methods (e.g., only about 1/22 of A-MEM and 1/3 of DC in AppWorld), as it avoids per-injection LLM judgment and additional experience management beyond the basic skill mechanism.

5.2 Main Results

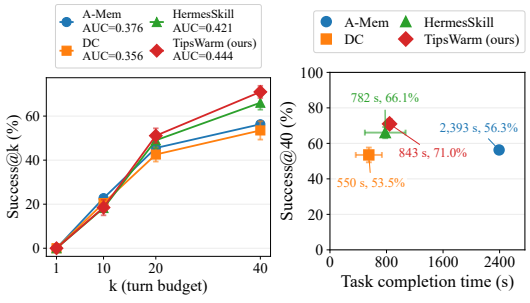
SkillsBench. Fig. 4 shows that TipsWarm improves both success rate and time cost efficiency on coding tasks requiring iterative debugging and testing. Compared with HermesSkill, TipsWarm not only achieves a higher success rate within 60 turns (84.1% vs. 82.3%) but also shows an advantage as early as 10 turns (30.4% vs. 22.1%), with AUC increasing from 0.456 to 0.533. Meanwhile, TipsWarm reduces the task completion time by 30.8%. These suggest that TipsWarm helps agents avoid unproductive attempts by keeping tips about host-grading lessons, with examples shown in Appendix D.1, and that the added tip-management cost can be offset by more efficient task execution. Against general memory baselines, TipsWarm achieves a final success rate that is 5.8 and 17.4 percentage points (pp) higher than those of A-MEM and DC, respectively, while also having the lowest completion time. A-MEM approaches TipsWarm's AUC but takes 5.6× as much time on successful tasks.



(a) Accuracy (b) Time Cost Efficiency

Figure 4: Performance on SkillsBench

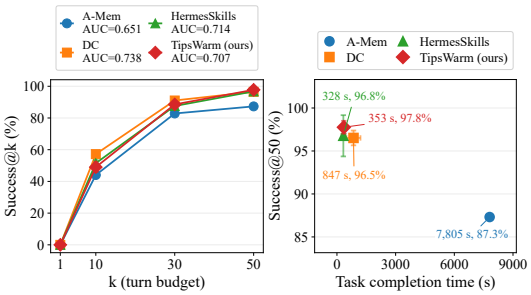
AppWorld. Fig. 5 shows that TipsWarm extends the accuracy benefits to coding tasks involving interactions with application APIs. Within 40 turns, TipsWarm achieves the highest success rate of 71.0%, exceeding HermesSkill and DC by 4.9 and 17.5 pp, respectively. Unlike on SkillsBench, this improvement comes with a modest increase (7.9%) in task completion time over HermesSkill. Nevertheless, TipsWarm remains 2.8× as fast as A-MEM, which requires memory retrieval and synchronization every few turns, but finally carries lots of task-specific experience fragments into the context, as we observe. As shown in Appendix D.2, TipsWarm can retain in the context the reminders to check API parameters and resolve identifiers, providing concise, reusable guidance for avoiding errors across different tasks.



(a) Accuracy (b) Time Cost Efficiency

Figure 5: Performance on AppWorld

AlfWorld. Fig. 6 shows that TipsWarm can be significantly more time-efficient than the general memory baselines on tasks with repetitive action patterns while achieving a high final success rate. TipsWarm achieves 1.2 pp and 10.4 pp higher success rates than DC and A-MEM, respectively, with only about 2/5 and 1/22 as much task completion time. Though DC's AUC is slightly higher than that of TipsWarm (0.738 vs. 0.707), it comes at the cost of 2.4× longer test time and 58.2× longer warmup time (Appendix B.2). For tasks with simple, repetitive interaction, DC's memory management may account for a larger share of runtime, making TipsWarm's inexpensive per-turn screening particularly useful. When HermesSkill already achieves a high success rate of 96.8%, TipsWarm inherits its skill ability and further raises the rate by about 1 pp by adding tips about command format and anti-loop rules in the context (Appendix D.3).

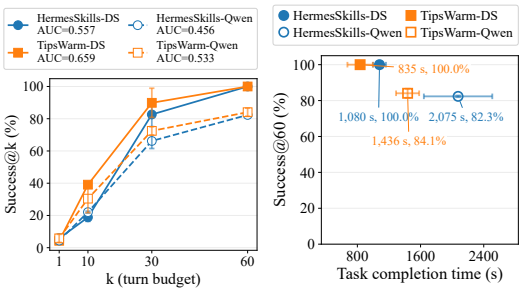


(a) Accuracy (b) Time Cost Efficiency

Figure 6: Performance on AlfWorld.

5.3 Ablation Study

Backbone LLM. We change the backbone LLM to DeepSeek-V4.1-Flash (DS) (DeepSeek-AI, 2026), a recently released Mixture-of-Experts (MoE) model with higher reasoning and coding capability. As Fig. 7 shows, for SkillsBench tasks, the stronger DS can directly increase HermesSkill's final success rate to 100% with an AUC of 0.557, corresponding to increases of 17.7 pp and 10.1 pp, respectively, over the results obtained with Qwen. TipsWarm not only has the potential to shrink the capability gap between LLM models (TipsWarm-Qwen AUC 0.533 vs. HermesSkill-DS AUC 0.557), but can also further extend the capability margin of stronger models, increasing HermesSkill-DS's AUC by another 10.2 pp to 0.659. This indicates that TipsWarm can help agents with strong LLMs complete complex coding tasks in fewer turns, which also explains its time cost efficiency improvement over HermesSkill.



(a) Accuracy (b) Time Cost Efficiency

Figure 7: Performance on SkillsBench using different backbone LLMs.

Mechanism. Table 1 shows TipsWarm's accuracy on SkillsBench across various settings. **NW**: the agent inherits HermesSkill's warmed-up skill bundle. **NS**: no scope matching during injection. **NU**: no utility attribution or screening. **BP**: a bigger pool containing at most 16 tips with up to eight tips injected per turn. **NONE** and **ALL** are the default HermesSkill and TipsWarm. As expected, scope matching and utility attribution help the agent to follow earlier guidance (NS vs. ALL and NU vs. ALL). A bigger pool can increase the final success rate but does not necessarily improve early-stage performance (BP vs. ALL). Excitingly, TipsWarm can augment an existing skill bundle without requiring prior TipsWarm-style warmup in the first place (NONE vs. NW vs. ALL).

Table 1: SkillsBench ablation results

Setting	Warmup w/ Tips	Scope Matching	Utility Attribution	Bigger Pool	Success@60 ↑ (%)	AUC ↑
NONE					82.3 ± 0.5	0.456 ± 0.034
NW		✓	✓		$\underline{85.5 \pm 2.5}$	$\mathbf{0.551 \pm 0.026}$
NS	✓		✓		84.1 ± 2.5	0.513 ± 0.059
NU	✓	✓			82.6 ± 0.0	0.454 ± 0.071
BP	✓	✓	✓	✓	$\mathbf{88.4 \pm 2.5}$	0.501 ± 0.028
ALL	✓	✓	✓		84.1 ± 2.5	$\underline{0.533 \pm 0.023}$

6 Conclusion

We presented TipsWarm to complements on-demand skill retrieval with warm tips selected for each conversation turn. By separating event-triggered LLM assessment from fast per-turn screening, it makes skill guidance readily available while controlling context and maintenance costs. Extensive evaluations on coding and iterative task-execution benchmarks show improved task success and success-time trade-offs over the underlying skill mechanism and general memory baselines.

AI USE STATEMENT

In this work, we used generative AI tools for implementing methods and providing feedback on experiments. Additionally, we used generative AI tools for correcting grammar errors and improving readability. We have *NOT* used generative AI tools to design or provide feedback on research methodology, assist with translation, clean and reformat datasets, support qualitative and thematic data analysis, or interpret results. Generating synthetic datasets, helping develop theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, and proposing or refining hypotheses are *NOT applicable* to this work. We have reviewed all AI-assisted work. For example, LLM-generated code was verified and tested for correctness. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We have taken steps to improve the reproducibility of our experiments. Section 4 describes the method, and Section 5.1 describes benchmark splits, baselines, evaluation metrics, and execution setup. Appendix A provides the LLM assessment prompts, and Appendix B summarizes the main hyperparameters and computational costs. The source code and experimental results are provided in the supplementary materials for anonymous review, which is indicated at the footnote in page 1. The code documentation provides detailed instructions for (1) preparing the environment, (2) preparing benchmarks and data, (3) configuring baseline methods, (4) running experiments, and (5) locating the archived result summaries for the experiments reported in this paper.

REFERENCES

Nous hermes agent, 2026. URL `https://github.com/nousresearch/hermes-agent`.

Qwen3.6-27b, 2026. URL `https://huggingface.co/Qwen/Qwen3.6-27B`.

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. Evoskill: Automated skill discovery for multi-agent systems. *arXiv preprint arXiv:2603.02766*, 2026.

DeepSeek-AI. Deepseek-v4.1-flash: Pushing the limits of kv cache compression, 2026.

Guanyu Jiang, Zhaochen Su, Xiaoye Qu, and Yi R Fung. Xskill: Continual learning from experience and skills in multimodal agents. *arXiv preprint arXiv:2603.12056*, 2026.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 13358–13376, 2023.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. volume 33, pp. 9459–9474, 2020.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026.

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

Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. Skillclaw: Let skills evolve collectively with agentic evolver. *arXiv preprint arXiv:2604.08377*, 2026.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. volume 36, pp. 8634–8652, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. {ALFW}orld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=0IOX0YcCdTn.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, Rabat, Morocco, March 2026. Association for Computational Linguistics.

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 *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024.

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

Hao Wang, Guozhi Wang, Han Xiao, Yufeng Zhou, Yue Pan, Jichao Wang, Ke Xu, Yafei Wen, Xiaohu Ruan, Xiaoxin Chen, et al. Skill-sd: Skill-conditioned self-distillation for multi-turn llm agents. *arXiv preprint arXiv:2604.10674*, 2026.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pp. 63897–63911. PMLR, 2025.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, et al. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 17577–17604. Curran Associates, Inc., 2025.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. volume 37, pp. 50528–50652, 2024.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, et al. Autoskill: Experience-driven lifelong learning via skill self-evolution. *arXiv preprint arXiv:2603.01145*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations*, pp. 9965–10017, 2025.

Danyang Zhang, Lu Chen, Situo Zhang, Hongshen Xu, Zihan Zhao, and Kai Yu. Large language models are semi-parametric reinforcement learning agents. volume 36, pp. 78227–78239, 2023.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, et al. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, XiongHui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, pp. 34040–34077, 2025.

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*, volume 38, pp. 19632–19642, 2024.

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*, pp. 19036–19066, 2024.

Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, et al. Memento-skills: Let agents design agents. *arXiv preprint arXiv:2603.18743*, 2026.

APPENDIX CONTENTS

A Prompts for LLM-based Assessment 13
B Experiment Setting Details 16
B.1 Hyperparameters 16
B.2 Running Cost 16
C Results of Token-Cost Efficiency 17
D Examples of Warm Tips 17
D.1 SkillsBench Tips 17
D.2 AppWorld Tips 18
D.3 AlfWorld Tips 18

A PROMPTS FOR LLM-BASED ASSESSMENT

This section lists three prompts used for LLM-based assessment in the filtering and attribution operations. The prompts are only used by a separate LLM to maintain the warm-tip pool but are not directly reflected in the conversation context for solving tasks.

Admission judge. Every time there is a skill event, including viewing, generating, and updating skills, the filtering operation uses the following prompt to admit candidate warm tips.

The Prompt for Admission Judge

You gate admission of individual hot-skill keypoints to a broadcast pool.
Every admitted tip will be injected on later tasks, including held-out tasks never seen while the pool was built. Admit only tips that remain meaningful after stripping instance-specific tokens and would help on an unseen task in the same benchmark/environment genre.
Return JSON only: {"admit": ["id", ...]} listing ids to admit from the provided candidates. Omit ids for non-transferable tips. Prefer admitting fewer strong tips over keeping marginal ones.
ADMIT tips that:

- State evaluator-/environment-agnostic process rules (write graded artifacts where the harness reads them; verify on the graded surface; use admissible APIs/tools). That is not a default env closer call
- Describe reusable pitfalls without absolute paths, task IDs, or oracle shortcuts

REJECT tips that:

- Tell the agent to run bundled solution/ scripts, read ground truth from tests, or skip builds using static expected constants
- Encode one episode's Docker/Lean/install recipe rather than a reusable rule
- Prescribe a termination ritual (always call done()/complete_task, always return a short/minimal answer). Those are episode procedures
- Prescribe a process ritual (ALWAYS show_api_doc / check docs before every call) rather than a reusable constraint
- Would mislead on an unrelated later task

Eviction judge. When the warm-tip pool lacks capacity for new candidates in the filtering operation, the following prompt determines which tips to evict from or retain in the pool.

The Prompt for Eviction Judge to Reconcile

You curate a small hot-skill key-point pool used as short guardrail reminders.
Hard constraint — keep is the store, not a guaranteed inject: every tip you KEEP may be prepended on later turns and on later tasks, including held-out tasks that were never seen while the pool was built. Inject later selects a matching subset. Optimize for unseen-task transfer and broadcast-safety (still helpful or at least harmless out of domain). Do NOT optimize for replaying the same tasks that produced the tips, and do NOT maximize usefulness on the present pool alone.
Return JSON only: {"keep": ["id", ...]} with at most {keep_n} ids drawn from the provided points. Do not invent ids. Prefer fewer strong tips over filling the quota: return an empty or short keep list when most candidates fail the bar.
KEEP tips that:

- State a transferable pitfall or procedure that remains meaningful after stripping instance-specific tokens (paths, hostnames, accounts, filenames, task IDs, one-off product recipes) and would still apply to a new task instance in the same environment/genre
- Encode evaluator-/environment-agnostic process rules (e.g. write graded artifacts where the harness reads them; prefer admissible actions; use the provided API surface; verify before long runs) rather than a single task's setup recipe or a default env closer
- Add distinct failure-mode coverage — prefer diversity of pitfalls across skills/topics when candidates are similar
- Use each point's natural-language scope (when present) as a soft hint. Prefer tips that are either broadly safe or clearly conditional; do not keep a narrow niche tip merely because it is useful on the current overflow
- When utility stats are present (n_labeled $>$ 0), prefer tips with more helpful than harmful attributions; when success rates are similar, prefer lower avg_iterations_when_success / avg_iterations (efficiency often separates tips when success is near-ceiling)

DROP or deprioritize tips that:

- Are bound to absolute paths, hostnames, credentials, account names, phone numbers, one-off filenames, task IDs, or a single product/CLI/ app workflow (unless the same sentence also states a general rule that survives removing those tokens)
- Recap one episode's entities or layout (which object was where, which inbox/account/file the last task used, numbered instance slots, exact names that will not recur). Identifier placeholders such as <path> or <id> do not make a recap transferable — if the remaining claim is still a memory of one episode, drop it
- Teach oracle/leakage or harness-cheating shortcuts (reading hidden solvers/oracles, claiming pass without producing graded outputs, rewriting verifier paths to temporary locations as a default)
- Encode one task's I/O layout, auth bootstrap, or dependency install recipe rather than a reusable pitfall
- Act mainly as a cheatsheet for replaying an already-seen task (memorized steps, exact entities, or solutions that would not help a new unseen task in the same benchmark)
- Duplicate a stronger tip already being kept, or add a second tip from the same skill that covers the same failure mode
- Would mislead, waste steps, or pull work off the graded surface if shown on an unrelated later task
- Prescribe a default search order over instance-indexed slots (visit every shelf / drawer / file / message N) rather than a constraint that applies only after the current observation fails. Keep a legal-action rule if it can stand without the tour
- Prescribe a termination ritual (always call done()/complete_task, always return a short/minimal answer). Those are episode procedures; the correct closer depends on the task. A NEVER/DO NOT about a bad closer is a constraint — keep that

- Have clearly worse utility than alternatives (high harmful count, or much higher iteration cost for similar success)

Imperative intensity (NEVER/ALWAYS/MUST) is not evidence of quality — judge the abstract claim, not the wording.
About context: it describes the incoming extract / current task only (often from the training task). Use it to understand what is being admitted and to break ties. Do NOT treat "most relevant to context", "will help if this task is repeated", or "likely needed on similar seen tasks" as the primary keep criterion under store-then-retrieve inject.
When utilities_flat_no_helpful is true in the user payload, no tip yet has a helpful attribution — prefer dropping oracle/cheatsheet candidates and episode-local recipes even if they are merely irrelevant.

Outcome-attribution judge. After task completion, the attribution operation uses the following prompt for LLM assessment to update tips' utility counters based on their contribution to the task outcome.

The Prompt for Outcome Attribution Judge

You attribute credit for task outcome to hot-skill tips that were exposed during the episode (injected or opened via skill_view).
The outcome is multi-dimensional — do NOT use success alone. Consider reward, iterations/steps (efficiency; lower is better when the task succeeded), tests_passed/tests_total, and duration when present. On benchmarks where success is near-ceiling, prefer judging whether a tip helped or hurt efficiency and reliability.
episode_log is a compressed action trace (tool names + short args, env actions, or code API snippets). It does NOT include observations or raw API dumps. Use it to see whether a tip was followed, ignored, or contradicted. A tip whose advice never appears in the log is irrelevant even on success. A tip that matches a long enumeration of lookups or extra env steps is harmful when iterations are high. Missing log fields are not evidence the tip helped.
When outcome.success is false OR outcome.claimed_success_mismatch is true: you MUST NOT label any via=inject tip as helpful. Prefer harmful for injected tips (they were broadcast into a failed episode). History-only tips may be irrelevant.
When outcome.claimed_success_mismatch is true, the agent asserted pass/completion in its final response but labeled evaluation failed — label injected tips that plausibly steered toward wrong-env verification, oracle/replay shortcuts, or premature success claims as harmful.
Return JSON only: {"labels": {"<id>": "helpful"|"harmful"| "irrelevant", ...}} for the given point ids. Do not invent ids.

- helpful: tip is a transferable process or constraint that plausibly improved this episode and would still apply to a new task instance
- harmful: tip plausibly caused waste, wrong paths, or failure modes, or would mislead on a different instance (wrong object, path, account). On task failure, prefer harmful over irrelevant for injected tips that prescribe shortcuts, oracle replay, or environment-specific cheats
- irrelevant: tip did not meaningfully affect this episode, OR it is an episode-local recap (specific objects/locations/accounts/filenames) even if the episode succeeded — those do not transfer

Judge the abstract claim, not wording. NEVER/ALWAYS/MUST is not evidence of helpfulness. Prefer irrelevant over helpful when a tip only restates one episode's entities. If a tip plausibly caused extra iterations — an enumeration tour, or visiting slots the observation did not suggest — label it harmful even when the episode eventually succeeded.

B EXPERIMENT SETTING DETAILS

B.1 HYPERPARAMETERS

Table 2 summarizes the default hyperparameters for running TipsWarm.

Table 2: Default hyperparameters for running TipsWarm

Hyperparameter	**Value**	**Description**
TipsWarm mechanism		
Pool size budget	12	Maximum number of candidate tips retained in the persistent pool.
Per-skill cap	8	Maximum tips admitted from any single skill, to avoid one skill dominating the pool.
Injection quota	4	Maximum tips inserted into the context of each user-message turn.
Scope match	lexical overlap	Exclude tips whose scope keywords do not overlap the task query.
Utility score	$+$helpful, $-$harmful/irrelevant	Rank remaining tips from attribution counts. Higher scores are injected first.
Warmup and test		
Backbone LLM	Qwen3.6-27B	For both agent and warm-tip judges, with reasoning and tool calling enabled.
Context length	131,072	Maximum context window of the deployed model.
Turn budget k	60 / 50 / 40	Maximum agent turns on SkillsBench / AlfWorld / AppWorld.
Protocol	warmup then checkpoint	The environment is saved after warmup and reused for the test split.
Repeats	3	Independent runs with mean $\pm$ std reported (A-MEM is run once).

B.2 RUNNING COST

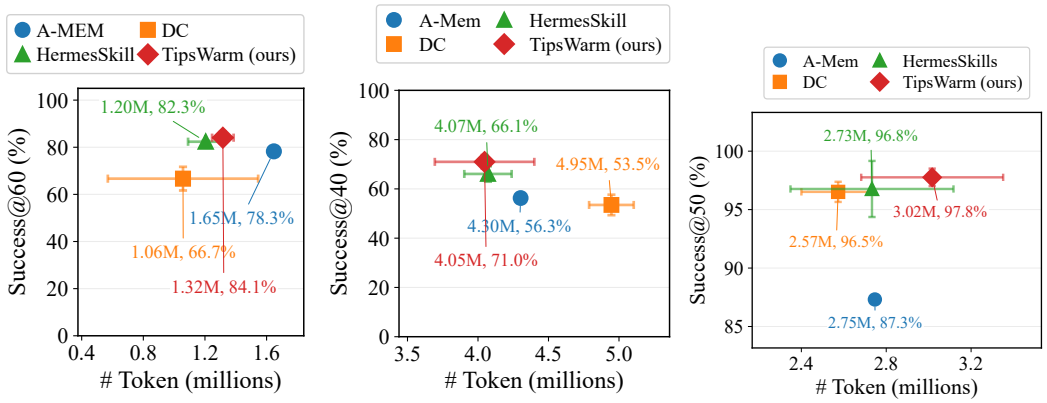
Table 3 summarizes the token, time, and space cost for different methods to initialize the skills or experience with the warmup tasks of various benchmarks. While the number of tokens varies across different methods and the storage space cost appears to be a minor practical concern, TipsWarm requires substantially less running time than the general memory methods.

Table 3: Training cost of different baselines for completing warmup tasks of various benchmarks. The number of agent tokens is the per-task mean $\pm$ std of tokens required by the agent to solve tasks, while the number of internal tokens is the number of tokens required by the baseline methods' internal operations. The time column reports the wall-clock duration required for the agent to finish the warmup tasks. Experience space is the persistent tip/memory artifact in addition to the default skills written by each method.

Benchmark	**Method**	N	**# Agent Tokens**	**# Internal Tokens**	**Time**	**Experience Space**
SkillsBench	A-MEM	65	1.74±1.69M	3.1K	316.0 h	2.2 MB: 65 notes, 115 KB; 2.1 MB sqlite3 data
	DC	65	2.24±1.78M	3.2K	134.4 h	84.7 KB: 5 cheatsheet items, 6.3 KB: 2 skills, 78.4 KB
	HermesSkill	65	2.53±1.65M	—	113.1 h	93.8 KB: 10 skills
	TipsWarm (ours)	65	2.47±2.05M	2.8K	28.6 h	105 KB: 2 skills, 103 KB; 2 tips, 1.7 KB
AppWorld	A-MEM	90	3.79±3.54M	48.9K	261.5 h	33.7 MB: 1482 notes, 2.5 MB; 31.2 MB sqlite3 data
	DC	90	8.24±14.39M	4.2K	40.1 h	20.2 KB: 4 cheatsheet items, 8.5 KB, 1 skill, 11.7 KB
	HermesSkill	90	3.74±3.33M	—	47.3 h	15.7 KB: 1 skill
	TipsWarm (ours)	90	3.91±3.05M	1.8K	12.2 h	73.3 KB: 1 skill, 71.1 KB; 3 tips, 2.2 KB
AlfWorld	A-MEM	50	0.67±0.76M	93.0K	76.9 h	37.4 MB: 1763 notes, 2.5 MB; 34.9 MB sqlite3
	DC	50	0.70±0.90M	13.1K	87.4 h	16.2 KB: 7 cheatsheet items, 9.3 KB; 1 skill, 6.9 KB
	HermesSkill	50	0.54±0.51M	—	1.5 h	3.1 KB: 1 skill
	TipsWarm (ours)	50	0.65±0.80M	1.4K	1.5 h	6.5 KB: 1 skill, 3.6 KB; 6 tips, 3.0 KB

C Results of Token-Cost Efficiency

Fig. 8 shows the success rate within a turn budget and the corresponding number of tokens consumed by agents using different methods on various benchmarks. Note that these agent tokens do not include tokens that may be required by methods' internal operations. No method consumes consistently fewer agent tokens than the others across all benchmarks.



(a) SkillsBench (b) AppWorld (c) AlfWorld

Figure 8: Token-cost efficiency performance on various benchmarks

D Examples of Warm Tips

We show examples of candidate warm tips remaining in the pool and their utility counters after all tasks are completed for different benchmarks. The full content in the warm-tip pool also includes other descriptive metadata for tips, such as their scope, tag, and injection-related statistics, which we choose to omit here for brevity. The quota of candidate tips that can enter the context is limited to four in the experiments.

D.1 SkillsBench Tips

Table 4 shows a snapshot of the warm-tip pool after a test run on the SkillsBench benchmark. The pool contains five candidate tips belonging to a single skill. Four of the five tips have a small positive utility (two helpful labels, no harmful or irrelevant labels), and the remaining tip is unlabeled. The final counters are sparse mainly for two reasons. One is the scope screening of the injection operation, and the other is that when any tip text is updated during reconciliation in the filtering operation, the tip is considered a new tip and its counters are reset. The helpful tips capture key requirements for host grading and artifact staging. Without this guidance in advance, agents may incur substantial testing and verification overhead.

Table 4: An example of candidate warm tips in the pool in a SkillsBench run

Tip	#Helpful	#Harmful	#Irrelevant
Skill: `skillsbench-host-grading`			
Graded files must be top-level task-dir entries (stager maps `/root`)	2	0	0
Mirror `/root/output/<f>` as top-level `output/<f>`	2	0	0
Do not reuse image-shipped `environment/` answers	2	0	0
Host pass needs `tests_total>0` and `tests_failed=0`	0	0	0
Stage `/root/results.json` as top-level `results.json`	2	0	0

D.2 APPWORLD TIPS

Table 5 shows the snapshot of the warm-tip pool after a test run on the AppWorld benchmark. The pool contains eight candidate tips belonging to two skills, also with sparse utility counters. The helpful tips focus mainly on API usage and identifier resolution.

Table 5: An example of candidate warm tips in the pool in an AppWorld run

Tip	#Helpful	#Harmful	#Irrelevant
Skill: `multi-app-api-workflow`			
Do not assume parameter names; check the API spec	3	2	0
Start from `show_app_descriptions()`	0	0	0
Never guess IDs; resolve them by search	0	0	0
Always call `show_api_doc` before use	0	0	0
Skill: `api-sandbox-tasks`			
Tokens expire across code blocks (401 if reused)	0	0	0
Never invent IDs, tokens, or passwords	3	2	0
Paginate until the page is empty	0	0	0
Standard library only; no external clients	0	0	0

D.3 ALFWORLD TIPS

Table 6 shows the snapshot of the warm-tip pool after a test run on the AlfWorld benchmark. The pool contains eight candidate tips belonging to one skill. These tips focus mainly on command format, action prerequisites, and avoiding repeated actions. All tips have accumulated helpful labels, and none has been labeled harmful. Each tip also has irrelevant labels, indicating that its usefulness varies across tasks. These short reminders provide guidance applicable across household tasks with little additional context.

Table 6: An example of candidate warm tips in the pool in an AlfWorld run

Tip	#Helpful	#Harmful	#Irrelevant
Skill: `alfworld-player`			
Use only commands from the admissible list	84	0	32
Reply with the bare command, with no quotes or prose	84	0	32
Find an object before trying to use it	84	0	32
Do not repeat a clean sequence that already finished	33	0	25
Stop after the goal is done; do not look or explain	11	0	12
Do not reopen a container just to check the object	33	0	23
Stop if the same command is repeated three times	84	0	32
Stop if open and close keep alternating	84	0	32