UNIFYING TEXT MEMORY AND CODE MEMORY FOR SELF-EVOLVING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

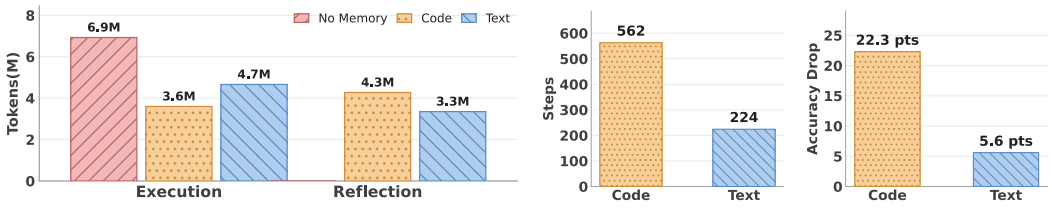
Self-evolving agents improve over time by distilling experience from past executions to facilitate future tasks. Existing systems represent such experience either as natural-language text injected into the agent context or as code that can be invoked via tools. However, the trade-offs between the two representations are poorly understood, and the choice between them is made a priori at the system level rather than based on the characteristics of each individual experience. We present the first controlled study that compares text memory and code memory under an identical task stream. The results show that the two memory forms are complementary in terms of construction cost, execution efficiency, and transfer robustness, such that neither representation alone is sufficient. Guided by these findings, we propose Metis, a self-evolving agent system that unifies text memory and code memory in a hierarchical manner. In particular, Metis organizes textual experience into execution plans, environment facts, and common pitfalls, and selectively crystallizes recurring plans into callable code tools. This design combines the good generality of text memory with the high execution efficiency of code memory and ensures that tool-generation cost is justified by repeated reuse. We evaluate Metis on three challenging benchmarks for interactive agents. Compared with a no-memory agent, SOTA baselines improve average accuracy by 9.5%–15.3%, while Metis improves accuracy by 26.9% and matches the most efficient baseline in execution cost.

1 INTRODUCTION

Large language models (LLMs) have evolved from passive text generators to agentic systems (Wang et al., 2024a) that interleave reasoning with action (Yao et al., 2023) and invoke external tools (Schick et al., 2023; Luo et al., 2025) to accomplish user-specified tasks. Once deployed, however, agents face open-ended environments whose challenges cannot be fully anticipated at training time, so good performance requires continual adaptation during interaction (ang Gao et al., 2025; Zheng et al., 2026; Huang et al., 2025). A fundamental obstacle is that LLM-based agents are stateless: experience gained during execution is confined to a finite context window and lost once the interaction ends, so when solving related tasks, an agent may rediscover known solutions or repeat earlier mistakes (Shinn et al., 2023; Zhao et al., 2024; Xiong et al., 2025). To tackle this problem, a natural solution is to preserve useful experiences beyond a single interaction. Self-evolving agents follow this paradigm by distilling experiences from past executions into persistent memory and reusing them in subsequent tasks (Zhang et al., 2026a; Cao et al., 2026; Li et al., 2026; Wu et al., 2026).

Existing approaches represent experiences in two primary forms, which we distinguish by how the agents consumes the stored experience rather than by the content of the experience. The first is *text memory*, where experiences are stored as natural-language knowledge and injected into the agent context at runtime, requiring the agent to read and reason over them (Zhang et al., 2026b; Han et al., 2025; Mi et al., 2026; Suzgun et al., 2025; Zhang et al., 2026a; Zhou et al., 2025). The second is *code memory*, where past routines are encapsulated as callable tools or MCP services that the agent invokes directly without re-deriving the underlying procedure (Li et al., 2026; Qiu et al., 2025; Zheng et al., 2025; Wang et al., 2025b). Under this criterion, systems that store code snippets but deliver them via context injection (Wang et al., 2026; Jiang et al., 2026) fall into the text-memory category, as the agent still consumes experiences by reading and reasoning them rather than by direct invocation.

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.



(a) Cost breakdown (b) Reflection steps (c) Transfer robustness

Figure 1: Comparing code memory and text memory under an identical task stream.

Despite that existing systems generate and manage experience memories in different ways, they choose between text or code memory a priori as ad hoc design decisions. The trade-offs between text and code memory thus remain insufficiently understood. It is unclear whether the two representations can be combined and which experiences should remain as text, which experiences should be codified.

To fill this gap, we conduct a controlled study on AppWorld (Trivedi et al., 2024), a stateful interactive-agent benchmark for multi-step API interactions. To check the influence of memory representation, we build text memory and code memory over the same task stream in the same order. The detailed settings of the profiling experiment are provided in Appendix A. The results reveal that the text memory and code memory exhibit complementary characteristics along three dimensions.

1) Construction Cost. The right plot of Figure 1a shows that code memory incurs higher construction token cost than text memory. Figure 1b also shows that constructing code memory takes more agent steps than text memory since additional exploration, validation, and debugging are required to ensure that the proposed implementation behaves correctly.

2) Execution Efficiency. When solving new tasks, the left plot of Figure 1a shows that code memory is more efficient than text memory. Text memory describes a routine in natural language, and thus the agent must reason over the memory. In comparison, code memory encapsulates the routine as a callable tool, allowing a multi-step procedure to be replaced by a direct invocation.

3) Transfer Robustness. Text memory transfers more reliably to unseen tasks than code memory. We illustrate by comparing an *Oracle* setting, where memory is constructed and evaluated on the same set of tasks, with a *streaming* setting, where each task can only use memory distilled from earlier tasks. The accuracy drop between the two settings captures how well a memory representation transfers beyond the tasks from which it was derived. Figure 1c shows that code memory has much higher transfer loss than text memory. This is because code memory commits experience into fixed executable behavior, which is efficient but less adaptable to task variation, whereas text memory remains natural-language guidance that can be reinterpreted during reasoning.

Our findings suggest that text memory and code memory are complementary. In particular, text memory is cheap to construct and robust under transfer, making it suitable for broadly capturing reusable experience. Code memory, in contrast, offers higher execution efficiency but is costly to construct and less reliable under task variation. Therefore, codification is only worthwhile when an experience has enough occurrences to exhibit a stable execution pattern that can be safely crystallized into code and indicates sufficient future reuse opportunities to amortize the construction cost. This motivates a selective promotion mechanism, where text memory first serves as a low-cost staging layer, and only recurrent, high-value, and stable experiences are promoted into code memory.

We thus propose Metis, a hierarchical memory system that combines text and code representations. Text memory forms the foundation and is organized into execution plans that summarize reusable routines, environment facts that capture environment-specific constraints, and pitfalls that prevent repeated mistakes. Only plans that recur across tasks are promoted into callable tools, so tool construction cost is amortized over repeated use and each tool is distilled from multiple task instances rather than a single trajectory. Facts and pitfalls remain textual, as they do not prescribe procedures. By combining text memories with selectively crystallized executable tools, Metis preserves the low construction cost and high transfer robustness of text memory while leveraging the execution efficiency of code memory.

We evaluate Metis on three interactive-agent benchmarks with distinct task styles and compare with SOTA baselines that utilize either text or code memory. Each baseline has a benchmark where its gain over the no-memory agent shrinks to 2.0–3.9 points, whereas Metis achieves the highest accuracy on all three and gains at least 8.3 points on each. Averaged over benchmarks, its relative gain is 26.9%, versus 9.5%–15.3% for the baselines, and it executes more efficiently than the most accurate baseline on each benchmark. Ablations further prove our design.

To summarize, this paper makes the following contributions:

- We present, to the best of our knowledge, the first controlled study of experience representation that compares text memory and code memory under an identical task stream and a shared agent backbone, disentangling the effect of memory representation from the underlying agent system.
- We introduce Metis, a dual-representation memory system grounded in the insights of our representation study. Metis treats text memory as the foundation and selectively promotes recurring plans into callable code, while retaining environment facts and common pitfalls as textual guidance.
- We evaluate Metis comprehensively and compare with SOTA baselines. The results show that Metis achieves high accuracy, runs efficiently, and generalizes across benchmarks.

2 PRELIMINARIES

Agentic Workflow. We consider a paradigm that is commonly used in the industry, where an agent solves a user task by interacting with a stateful sandbox environment through ReAct loop (Yao et al., 2023; 2024; Yang et al., 2024). A task query given by the user is a natural-language instruction q, and the agent is an LLM policy π acting over an interaction context c_t (c_1 holds q and a system prompt). At step t the policy produces an action $a_t \sim \pi(\cdot \mid c_t)$, either a code block or a tool call, treated uniformly since both are text the system parses then executes (Wang et al., 2024b; Schick et al., 2023), and the environment returns an observation o_t, extending the context as $c_{t+1} = c_t \oplus (a_t, o_t)$. The interaction halts at step T, producing a trajectory: $\tau = \big(q,\ (a_1, o_1),\ \ldots,\ (a_T, o_T)\big)$.

Self-Evolving Agent Systems. Self-evolving agents improve over time by incorporating experience acquired from past task executions. Existing approaches broadly follow two paradigms: updating model parameters from collected experience (Zhao et al., 2025; Wu et al., 2026; Wang et al., 2025a; Zhang et al., 2026c), or retaining experience as explicit memory while keeping the underlying policy frozen (Zhang et al., 2026a; Qiu et al., 2025; Li et al., 2026; Zhang et al., 2026b). We focus on the latter, training-free setting, where experience is stored as model-agnostic artifacts that can be reused during inference, as it makes it easier to transfer experience memories when the underlying model is replaced (Fang et al., 2026). Concretely, the agent faces a stream of tasks $q^{(1)}, q^{(2)}, \ldots$ and maintains a memory M, with $M_0 = \emptyset$. For task $q^{(k)}$, relevant memory entries are retrieved from M_{k-1} and incorporated into the agent context, yielding actions $a_t \sim \pi(\cdot \mid c_t; M_{k-1})$. After the rollout $\tau^{(k)}$, reflector modules distill reusable experience and update the memory, $M_k = \mathrm{Reflect}(M_{k-1}, \tau^{(k)})$. This execute–reflect–reuse loop underlies a broad family of self-evolving agent systems (Han et al., 2025; Liu et al., 2025).

These systems differ primarily in how the retained memory is represented. Text-based approaches store experience as natural-language plans, strategies, playbooks, or structured skill descriptions (Zhang et al., 2026b; Zhou et al., 2025; Suzgun et al., 2025; Zhang et al., 2026a; Ouyang et al., 2026; Wang et al., 2026; Jiang et al., 2026; Mi et al., 2026). Code-based approaches instead materialize experience as executable tools or MCP services that can be invoked directly for future tasks (Li et al., 2026; Qiu et al., 2025; Wang et al., 2025b; Zheng et al., 2025). In most existing systems, however, the representation form is fixed *a priori* at the system level rather than selected according to the characteristics of each form. This representation-centric limitation is the focus of our work.

3 THE METIS SYSTEM

3.1 SYSTEM OVERVIEW

The overall architecture of Metis is illustrated in Figure 2. Metis maintains a hierarchical experience memory $M = (M_{\text{text}}, M_{\text{code}})$, consisting of a structured text memory store and a code memory library.

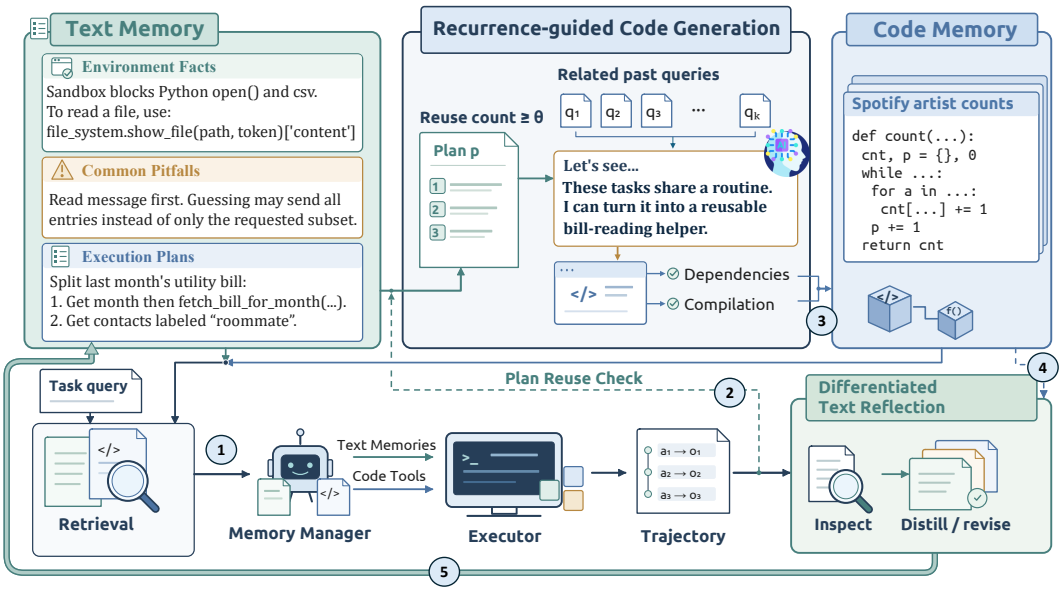


Figure 2: Architecture of Metis

Built upon this memory hierarchy, Metis operates through a self-evolving loop comprising three key components:

❶ *Differentiated Text Reflection* (§3.2) distills completed trajectories into three complementary forms of textual memories: execution plans, environment facts, and common pitfalls.

❷ *Recurrence-Guided Code Generation* (§3.3) selectively promotes recurring execution plans into callable tools rather than indiscriminately codifying experiences. When a plan has been reused sufficiently often, Metis treats it as evidence of a stable behavioral pattern and invokes a codifier to generate executable tools using the plan and related task queries.

❸ *Memory Manager* (§3.4) determines which memories should be reused for a new task. It first performs embedding-based top-k retrieval over both text memories and tool descriptions, and then employs an LLM-based manager to jointly select the most suitable entries.

Workflow. For each task, the *Memory Manager* selects relevant text entries and tools from M, and the executor solves the task with them. After execution, any selected plan with sufficient reuse evidence is first promoted into tools by *Recurrence-Guided Code Generation*. *Differentiated Text Reflection* then updates the text memory given the trajectory and the updated tool library.

3.2 Differentiated Text Reflection

Memory Schema. Metis organizes text memory into three categories, $M_{\text{text}} = M_{\text{env}} \cup M_{\text{pit}} \cup M_{\text{plan}}$. *Environment facts* ($M_{\text{env}}$) capture task-independent properties of the execution environment and its constraints. *Common pitfalls* (M_{pit}) describe structured failure patterns distilled from failed or inefficient executions, each represented as a *(trigger, mistake, consequence)* tuple. *Execution plans* (M_{plan}) describe reusable templates for solving recurring sub-routines. Unlike the other two categories, execution plans capture procedural patterns and are therefore the only type of textual experience eligible for promotion into code.

Formally, each entry $m = (\text{content}, \kappa, v) \in M_{\text{text}}$ contains a natural-language content, a category label $\kappa \in \{\text{env}, \text{pit}, \text{plan}\}$, and an invalidation bit $v \in \{0, 1\}$. Execution plans carry additional fields for codification (§3.3). For task i, Metis' Memory Manager(§3.4) selects $S_i = \textsc{Select}(q_i, M_{i-1})$, generate an execution trajectory $\tau_i \sim \pi(\cdot \mid q_i, S_i)$, and updates $M_i = \textsc{Reflect}(M_{i-1}, \tau_i)$.

Agentic Reflection. After receiving the executor's trajectory, the Text Memory Reflector distills it into reusable textual memories. It operates as an agent with a bounded reasoning loop and access to

the executor's tools and documentation, allowing it to gather evidence beyond the recorded trajectory. For failed tasks, it probes the environment to identify the underlying cause and proposes candidate entries suited to the failure type. For successful tasks, it still examines the trajectory for inefficiencies, such as erroneous retries or unnecessary exploration, and proposes efficiency-oriented entries, usually an execution plan, to avoid them.

To prevent memories from overfitting to a single task, the reflector is given the queries of the most recent N tasks. This allows it to observe recent query distribution, thus generating more general and reusable guidance that captures recurring patterns across tasks rather than task-specific solutions. A candidate entry is finalized only if the reflector judges it to be sufficiently general and it passes the remaining validation gates for non-redundancy, mechanistic grounding, and scope checking.

Memory Maintenance. The reflector performs not only memory creation but also memory maintenance through *merge* and *update* operations. When multiple narrow entries are found to capture the same underlying pattern, they are merged into a single, more general entry. When a previously distilled conclusion is found to be inaccurate, biased, or superseded by newer evidence, the corresponding entry is revised accordingly. Both operations follow an invalidate-and-replace policy rather than in-place editing. Specifically, each superseded entry has its invalidation bit flipped ($v : 0 \to 1$), and the merged or revised entry is appended as a new record. This design preserves memory consistency while ensuring that obsolete entries are excluded from future retrieval.

Memory Reuse. Selected entries are rendered as natural-language guidance and prepended to the executor's context. Since text acts as advice rather than control flow, the executor may adapt or ignore any entry, so an imperfect entry costs at most a disregarded suggestion rather than an error propagated through every invocation. This graceful degradation makes text the low-commitment default, while code is admitted only under the stricter conditions of §3.3.

3.3 Recurrence-guided Code Generation

Since code tools are costly to construct and may transfer poorly when over-specialized, Metis must address two questions before codification: when should tool generation be triggered, and what evidence should be provided to the codifier? First, a routine should be codified only after it has been observed frequently enough to justify the construction cost. Second, the codifier should observe multiple tasks involving the same routine, so that the generated tool captures their shared procedure rather than overfitting to a single task. Execution plans provide a natural basis for both. A plan describes a reusable procedure, and its repeated selection across tasks indicates that the procedure recurs in the workload. Meanwhile, the queries of these tasks provide different instances of the same procedure and expose the variations that the generated tool should support. Metis therefore uses execution plans both to determine when codification is triggered and to collect the task evidence used for tool generation. Accordingly, each plan $p \in M_{\text{plan}}$ augments the generic text-memory fields with three code-related fields:

$$p = \big(\text{body}(p), v(p), B(p), R(p), D(p)\big).$$

Here, $\text{body}(p)$ denotes the plan content and $v(p)$ its invalidation bit. $B(p)$ is a candidate buffer of recent tasks for which p was selected, $R(p)$ is a bounded history of related tasks retained under budget β, and $D(p)$ stores the admitted tools associated with the plan. $B(p)$ is used to detect repeated plan selection, while $R(p)$ preserves earlier task instances for subsequent codification rounds.

Recurrence Signal. Whenever a plan p is selected for task i, Metis records the task in its candidate buffer:

$$B(p) \leftarrow B(p) \cup \{i\}, \quad \text{if } p \in S_i^{\text{plan}},$$

where S_i^{plan} denotes the execution plans selected for task i. Once $|B(p)| \geq \theta$, Metis triggers codification for p, where θ is a configurable reuse threshold. The task need not succeed: plan selection itself indicates that the query matches the routine described by p and therefore contributes evidence about where the routine applies. When codification is triggered, the codifier receives the plan body, the candidate queries $Q_B(p) = \{q_i \mid i \in B(p)\}$, the related queries $Q_R(p) = \{q_i \mid i \in R(p)\}$, the current tool library rendered as signatures and docstrings, and access to the sandbox environment for inspection and execution testing.

Raw trajectories are deliberately excluded. Trajectories often contain exploratory calls, failed attempts, incidental variables, and task-specific constants that should not be carried into a reusable tool. Instead, the plan provides the procedure to implement, while $Q_B(p) \cup Q_R(p)$ provides the task variations that the implementation should support. This also allows unsuccessful tasks to contribute coverage through their queries without propagating erroneous execution paths into the generated tool. After the generated tools are successfully admitted, the consumed candidate tasks are folded into the bounded related-task history,

$$R(p) \leftarrow \mathrm{Recent}_\beta\big(R(p) \cup B(p)\big), \qquad B(p) \leftarrow \emptyset.$$

In this way, $B(p)$ tracks new recurrence since the previous codification round, while $R(p)$ retains earlier task instances so that subsequent codification can account for a broader set of observed uses. To support efficient reflection and reliable tool execution, Metis further employs a set of harness mechanisms for controlled evidence inspection, dependency handling, and tool validation. We defer these designs to Appendix C.

3.4 Memory Manager

Exposing all memory entries to the agent can introduce substantial noise, especially when many entries are irrelevant to the current task. Thus, Metis employs an LLM-powered Memory Manager to retrieve only the memory entries that are useful for the current task. For task i, Metis first performs embedding-based top-k retrieval over both text memory entries and code tools, using the query to construct a bounded candidate pool from each store. Memory Manager then jointly selects useful text memories and tools for the task, producing S_i^{text} and S_i^{tool}. The Manager is given the task query, the retrieved text entries, and the retrieved tool docstrings. This joint selection allows the Manager to reason over the interaction between textual guidance and executable artifacts. A selected plan can bring in the tools that implement it, while a selected tool can be paired with the text memory that describes when and why it should be used. The resulting selections are then passed to the dependency-checking pipeline (See Appendix C for details). For plans, $\mathrm{Ref}(p)$ includes the registered implementation set $D(p)$, so selecting a codified plan automatically makes its tools eligible for execution. We provide four case studies in Appendix E for a better understanding.

4 Experiments

4.1 Experiment Setup

Benchmarks. We evaluate Metis on three stateful interactive-agent benchmarks: AppWorld (Trivedi et al., 2024), OfficeBench (Wang et al., 2024c), and STATE-Bench (Microsoft, 2026). While all three benchmarks involve stateful interactions between agents and environments, they differ substantially in task characteristics. We analyze these differences when interpreting the main results. For AppWorld, we use the official training split (90 tasks) for memory construction and the test-normal split (168 tasks) for evaluation. For OfficeBench, we use 93 single-app tasks for memory construction and 207 multi-app tasks for evaluation. For each domain of STATE-Bench, we adopt the official training (100 tasks) and test (50 tasks) splits. The detailed description of the benchmarks are listed in Appendix D.

Models. We conduct the main experiments across two model families: DeepSeek-V4-Flash and Grok 4.6 for their outstanding agent abilities. All model temperatures are set as 1. For methods where embedding is used, we use text-embedding-3-large. For STATE-Bench, we consistently use DeepSeek-V4-Flash as user simulator and LLM judge.

Baselines. We compare Metis with a Base Agent that runs the ReAct executor without memory, and three self-evolving methods: ACE (Zhang et al., 2026a), which maintains experience as a natural-language playbook injected into the executor context. ASI (Wang et al., 2025b), which induces executable code tools from successful trajectories. And SkillX (Wang et al., 2026), which stores reusable knowledge, including code snippets, as structured skills injected into the executor context, placing it in the text-memory category under our criterion (§1). For fair comparison, we adapt all baselines to the same executor, task order, sandbox interface, and environment-awareness prompt, while preserving their original memory representations and update mechanisms based on their official implementations. Appendix B provides the configuration details and execution protocols.

Table 1: Main results of DeepSeek-V4-Flash. Accuracy changes are absolute differences relative to the corresponding Base Agent (No Memory), while executor-token and turn changes are relative percentages. Construction cost reports the total token cost of memory construction.

Benchmark	Metric	Base Agent	ACE	ASI	SkillX	Metis (Ours)
AppWorld	Acc. ↑	85.7%	88.7% +3.0	91.1% +5.4	92.3% +6.6	**94.0%** +8.3
	Avg. Exec. Tokens(K) ↓	183	1165 ↑ 536.6%	541 ↑ 195.6%	197 ↑ 7.7%	**188** ↑ 2.7%
	Turns ↓	17.5	21.1 ↑ 20.6%	11.1 ↓ 36.6%	16.4 ↓ 6.3%	**15.9** ↓ 9.1%
	Construction Cost(M) ↓	–	11.1	10.6	16.2	**8.5**
OfficeBench	Acc. ↑	30.4%	36.2% +5.8	41.5% +11.1	34.3% +3.9	**44.9%** +14.5
	Avg. Exec. Tokens(K) ↓	53	160 ↑ 201.9%	316 ↑ 496.2%	53 ↓ 0.0%	**68** ↑ 28.3%
	Turns ↓	15.7	14.2 ↓ 9.6%	11.9 ↓ 24.2%	14.6 ↓ 7.0%	**11.3** ↓ 28.0%
	Construction Cost(M) ↓	–	1.6	3.8	2.3	**6.7**
STATE-Bench	Acc. ↑	66.0%	80.7% +14.7	68.0% +2.0	71.3% +5.3	**81.3%** +15.3
	Avg. Exec. Tokens(K) ↓	36	367 ↑ 907.4%	541 ↑ 1384.9%	72 ↑ 97.8%	**67** ↑ 83.2%
	Turns ↓	6.8	7.1 ↑ 4.4%	7.8 ↑ 14.7%	7.1 ↑ 4.4%	**6.9** ↑ 1.5%
	Construction Cost(M) ↓	–	20.1	39.4	18.8	**29.9**

For Metis, we set the codification threshold to 3 and retrieval top-k to 10 for each memory category, resulting in 30 text memory entries and 10 code tool entries in total.

Evaluation Metrics. We evaluate each method along three dimensions: task accuracy, execution efficiency, and memory-construction cost. For task accuracy, we report the pass@1 task completion rate under each benchmark's official success criterion. For execution efficiency, we report the average number of ReAct turns and executor tokens (input plus output) per task. For methods that use LLM-based memory retrieval (e.g., the Memory Manager in Metis), executor tokens also include the retrieval cost. For construction cost, we report the total tokens consumed by each method's memory-construction procedure. Execution cost recurs on every task, whereas construction cost is paid once and amortized over subsequent reuse.

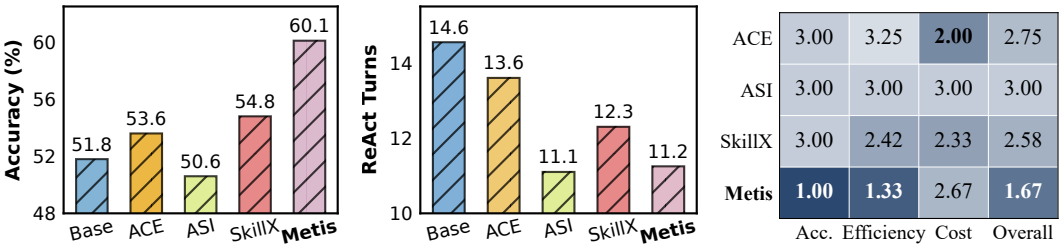
4.2 Main Result Analysis

Consistent Gains Across Task Regimes. Table 1 summarizes the results with DeepSeek-V4-Flash. Metis achieves the highest task accuracy on all three benchmarks, whereas the baselines exhibit distinct strong and weak regimes, consistent with the complementary strengths of the two memory representations: text memory offers flexible guidance on how to act, while code memory makes known routines more reliable and efficient to execute. The three benchmarks stress these strengths differently. OfficeBench usually outlines a coarse execution plan in the query, rewarding the reliable execution of known workflows that code memory provides. STATE-Bench leaves much of the procedure ambiguous, requiring the agent to determine what information to collect from the user and which constraints to satisfy, which makes flexible textual guidance particularly valuable. AppWorld lies in between, combining environment exploration with multi-step API composition. Consequently, no single baseline remains consistently strong across regimes: every baseline comes close to Metis on its favorable benchmark but falls far behind elsewhere. In terms of accuracy, ACE ranks first among the baselines on STATE-Bench but last on AppWorld. ASI ranks first on OfficeBench, where its tools also cut ReAct turns by 24.2%, but last on STATE-Bench. SkillX ranks first on AppWorld but last on OfficeBench: its hybrid skills pair strategic guidance with code snippets that suit AppWorld's mixed demands, but since the executor reads rather than invokes them, they provide little of the procedural compression that OfficeBench rewards.

Metis combines the strengths of both representations: textual guidance remains available when the agent must decide how to act, while procedures are compiled into tools once they recur often enough to justify codification. Beyond accuracy, Metis is also more execution-efficient than the highest-accuracy baseline on each benchmark, using fewer turns and executor tokens than SkillX on AppWorld, ASI on OfficeBench, and ACE on STATE-Bench. Figure 3c summarizes these trade-offs as average ranks across benchmarks. Metis ranks first in accuracy (1.00) and execution efficiency (1.33) while remaining competitive in construction cost (2.67), yielding the best overall rank (1.67 vs.

Table 2: Main results of Grok 4.6, in the same format as Table 1.

Benchmark	Metric	Base Agent	ACE	ASI	SkillX	Metis (Ours)
OfficeBench	Acc. ↑	30.4%	40.1%+9.7	38.6%+8.2	34.8%+4.4	**40.6%**+10.2
	Avg. Exec. Tokens(K) ↓	38	488↑ 1184.2%	315↑ 728.9%	44↑ 15.8%	**45**↑ 18.4%
	Turns ↓	14.2	15.1↑ 6.3%	11.0↓ 22.5%	12.8↓ 9.9%	**10.7**↓ 24.6%
	Construction Cost(M) ↓	–	4.3	1.6	3.6	**5.5**
STATE-Bench	Acc. ↑	67.3%	80.7%+13.4	72.0%+4.7	72.7%+5.4	**82.0%**+14.7
	Avg. Exec. Tokens(K) ↓	50	543↑ 986.0%	242↑ 384.0%	95↑ 90.0%	**66**↑ 32.0%
	Turns ↓	7.2	7.2↑ 0.0%	5.8↓ 19.4%	7.7↑ 6.9%	**6.7**↓ 6.9%
	Construction Cost(M) ↓	–	29.7	29.1	26.0	**19.0**



(a) Task accuracy (b) ReAct turns per task (c) Average rank

Figure 3: (a, b) Cross-model experience transfer on AppWorld, with memory constructed by Claude Sonnet 4.6 and executed by GPT-4o. (c) Average rank of the four self-evolving methods across the three benchmarks with DeepSeek-V4-Flash (lower is better). Methods are ranked per benchmark; Efficiency averages the executor-token and turn ranks, and Overall averages the three dimensions.

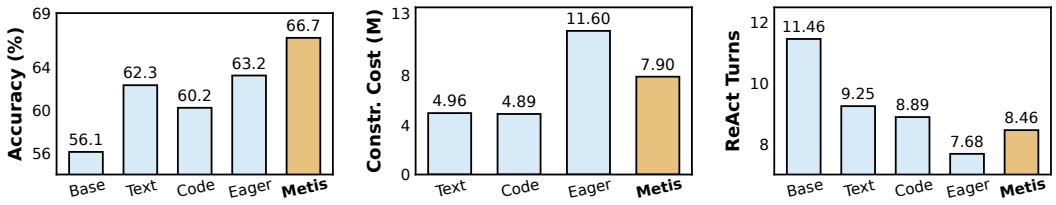
2.58–3.00 for the baselines). While individual baselines excel in particular regimes or dimensions, Metis remains consistently strong across all of them.

Effectiveness Across Model Families. We further evaluate Metis with Grok 4.6 to examine its robustness across model families. Since the Grok 4.6 base agent already approaches saturation on AppWorld, achieving 94.05% accuracy, we focus the Grok 4.6 experiments on OfficeBench and STATE-Bench, where self-evolving methods still have substantial room to improve performance. As shown in Table 2, Metis again achieves the highest accuracy on both benchmarks, improving the base agent by 10.2 points on OfficeBench and 14.7 points on STATE-Bench, while reducing ReAct turns by 24.6% and 6.9%, respectively. On OfficeBench, it achieves the largest accuracy gain while requiring the fewest turns. ACE is the strongest baseline under Grok 4.6 and comes within 1.3 points of Metis on both benchmarks, but consumes $10.8\times$ and $8.2\times$ the executor tokens of Metis. On STATE-Bench, Metis also outperforms ASI, which uses fewer turns, by 10.0 points in accuracy. These results suggest that the benefit of combining textual and executable experience is not specific to a specific model family.

Cross-Model Experience Transfer. We further examine whether experience constructed by a stronger model can be reused by a weaker executor on AppWorld. We use Claude Sonnet 4.6 for memory construction across all memory-based methods and GPT-4o as the executor, while keeping the remaining experimental setup unchanged. As shown in Figures 3a and 3b, except ASI, all methods transfer useful experience to GPT-4o to varying degrees. Metis yields the largest improvement in task accuracy while also reducing ReAct turns relative to the Base Agent, showing that it remains effective when memory construction and execution use different models.

4.3 Ablations

To disentangle the effects of individual design decisions, we conduct ablation studies to answer two questions: (1) Does the hybrid representation outperform either representation form alone? and (2) Does recurrence-gated promotion outperform eager per-task codification? All variants independently



(a) Task accuracy (b) Memory-construction cost (c) ReAct turns per task

Figure 4: Ablations on AppWorld. *Text* and *Code* each keep a single memory form. *Eager* keeps both forms but codifies after every task instead of using the recurrence gate.

construct memory on the official training split and are evaluated on the dev split following the standard protocol (Trivedi et al., 2024), using GPT-4o as the executor and Claude Sonnet 4.6 as the reflector.

Memory Forms. We first compare Metis with two single-representation variants. *Text-only* disables code generation, while *Code-only* disables text memory and directly constructs executable tools from individual tasks. As shown in Figure 4, both variants outperform No Memory, with gains of 6.2 and 4.1 points respectively. The full Metis further improves accuracy by 4.4 points over Text-only and 6.5 points over Code-only. These results show that neither representation alone captures the full benefit of experience reuse. Text memory retains broadly applicable guidance, while code memory efficiently executes recurring procedures. Metis combines the two by using execution plans as the basis for selective codification.

Promotion Strategies. We next examine whether recurrence is necessary for deciding when to codify experience. The *Eager* variant retains both memory forms but triggers code generation after every task. Compared with Metis, Eager variant increases construction cost by 47%, while reducing accuracy from 66.7% to 63.2%. It achieves slightly lower execution cost because more procedures are converted into tools, but many of these tools provide little future value. This suggests that recurrence provides a useful signal for distinguishing procedures worth codifying from task-specific patterns, reducing unnecessary construction cost while maintaining the execution benefits of code memory.

5 Conclusion

In this work, we revisit a design dimension that existing self-evolving agents fix a priori: the representation form of accumulated experience. Our controlled study shows that text and code memories exhibit complementary trade-offs, and our main experiments confirm the practical consequence: methods committed to a single representation each exhibit a task regime where their benefit largely vanishes. Metis maintains differentiated text memory as a low-commitment foundation and selectively promotes recurring plans into validated callable tools. Across three benchmarks, it is the only method that improves substantially over the no-memory agent in every regime, while achieving the best overall balance between task accuracy and execution efficiency. Our ablations attribute these gains to the hybrid representation and the recurrence gate respectively.

6 Limitations

Our evaluation only covers agentic task execution in application environments and leaves workloads such as coding and QA unexplored. Nonetheless, we expect the dual representation to take effect on their harness: recurring build, test, or retrieval routines can be promoted into tools, while conventions and reasoning strategies remain in text. Moreover, a flawed text entry can be revised by later reflection, whereas a codified tool bypasses LLM reasoning and replays a flawed routine until it is fixed. Sandbox testing before admission cannot rule out subtle errors, and API drift can invalidate tools that were once correct, so revising registered tools is an important direction for future work. Finally, we did not tune hyperparameters: a single configuration ($\theta = 3$, top-10 retrieval per memory category) achieves the highest accuracy in all five benchmark–model settings, so the gains of Metis do not rely on per-environment tuning.

7 AI USE STATEMENT

In this work, we used generative AI tools to provide feedback on research methodology or experiments, implement partial methods, and interpret results. We have not used generative AI tools to help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, assist with translation, or clean and reformat datasets. Meanwhile, generating synthetic data sets and supporting for qualitative and thematic data analysis are not applicable to this work. Additionally, we used generative AI tools for brainstorming, creating or editing software code, and editing a research paper to improve readability. We have reviewed all AI-assisted work. AI-assisted code was reviewed and tested by the authors before being incorporated into the implementation or experimental pipeline. AI-assisted feedback on experiments and interpretations of experimental results were independently evaluated by the authors against the underlying experimental evidence, and all final methodological and experimental decisions were made by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

8 REPRODUCIBILITY STATEMENT

Appendix B specifies the evaluation protocol and the configuration of every evaluated method, §4.1 specifies the models and dataset splits, and Appendix F gives the prompts of all Metis components.

REFERENCES

Huan ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, Hongru Wang, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Qihan Ren, Cheng Qian, Zhenghailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: On path to artificial super intelligence, 2025. URL https://arxiv.org/abs/2507.21046.

Zouying Cao, Jiaji Deng, Li Yu, Weikang Zhou, Zhaoyang Liu, Bolin Ding, and Hai Zhao. Remember me, refine me: A dynamic procedural memory framework for experience-driven agent evolution, 2026. URL https://arxiv.org/abs/2512.10696.

Gaodan Fang, Vatche Isahagian, K. R. Jayaram, Ritesh Kumar, Vinod Muthusamy, Punleuk Oum, and Gegi Thomas. Trajectory-informed memory generation for self-improving agent systems, 2026. URL https://arxiv.org/abs/2603.10600.

Dongge Han, Camille Couturier, Daniel Madrigal Diaz, Xuchao Zhang, Victor Rühle, and Saravan Rajmohan. Legomem: Modular procedural memory for multi-agent llm systems for workflow automation, 2025. URL https://arxiv.org/abs/2510.04851.

Yuchen Huang, Sijia Li, Zhiyuan Fan, Minghao LIU, Wei Liu, and Yi R. Fung. Scaling environments for LLM agents: Fundamentals, approaches, and future directions. In *Workshop on Scaling Environments for Agents*, 2025. URL https://openreview.net/forum?id=9axZcDTiJm.

Guanyu Jiang, Zhaochen Su, Xiaoye Qu, and Yi R. Fung. Xskill: Continual learning from experience and skills in multimodal agents, 2026. URL https://arxiv.org/abs/2603.12056.

Haotian Li, Shijun Yang, Weizhen Qi, Silei Zhao, Rui Hua, Mingzhu Song, Xiaojian Yang, and Chao Peng. Yunjue agent tech report: A fully reproducible, zero-start in-situ self-evolving agent system for open-ended tasks, 2026. URL https://arxiv.org/abs/2601.18226.

Jiarun Liu, Shiyue Xu, Yang Li, Shangkun Liu, Yongli Yu, and Peng Cao. Unifying dynamic tool creation and cross-task experience sharing through cognitive memory architecture, 2025. URL https://arxiv.org/abs/2512.11303.

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, 2025. URL https://arxiv.org/abs/2508.14704.

Qirui Mi, Zhijian Ma, Mengyue Yang, Haoxuan Li, Yisen Wang, Haifeng Zhang, and Jun Wang. Skill-pro: Learning reusable skills from experience via non-parametric ppo for llm agents, 2026. URL https://arxiv.org/abs/2602.01869.

Microsoft. STATE-Bench: Stateful task agent evaluation benchmark. https://github.com/microsoft/STATE-Bench, 2026. Accessed 2026-09-25.

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, 2026. URL https://arxiv.org/abs/2509.25140.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, Xing Zhou, Dongrui Liu, Ling Yang, Yue Wu, Kaixuan Huang, Shilong Liu, Hongru Wang, and Mengdi Wang. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution, 2025. URL https://arxiv.org/abs/2505.20286.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory, 2025. URL https://arxiv.org/abs/2504.07952.

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, 2024. URL https://arxiv.org/abs/2407.18901.

Chenxi Wang, Zhuoyun Yu, Xin Xie, Wuguannan Yao, Runnan Fang, Shuofei Qiao, Kexin Cao, Guozhou Zheng, Xiang Qi, Peng Zhang, and Shumin Deng. Skillx: Automatically constructing skill knowledge bases for agents, 2026. URL https://arxiv.org/abs/2604.04804.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), March 2024a. ISSN 2095-2236. doi: 10.1007/s11704-024-40231-1. URL http://dx.doi.org/10.1007/s11704-024-40231-1.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024b. URL https://arxiv.org/abs/2402.01030.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, Yiping Lu, Kyunghyun Cho, Jiajun Wu, Li Fei-Fei, Lijuan Wang, Yejin Choi, and Manling Li. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning, 2025a. URL https://arxiv.org/abs/2504.20073.

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, 2024c. URL https://arxiv.org/abs/2407.19056.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks, 2025b. URL https://arxiv.org/abs/2504.06821.

Rong Wu, Xiaoman Wang, Jianbiao Mei, Pinlong Cai, Daocheng Fu, Cheng Yang, Licheng Wen, Xuemeng Yang, Yufan Shen, Yuxin Wang, and Botian Shi. Evolver: Self-evolving llm agents through an experience-driven lifecycle, 2026. URL https://arxiv.org/abs/2510.16079.

Zidi Xiong, Yuping Lin, Wenya Xie, Pengfei He, Zirui Liu, Jiliang Tang, Himabindu Lakkaraju, and Zhen Xiang. How memory management impacts llm agents: An empirical study of experience-following behavior, 2025. URL https://arxiv.org/abs/2505.16067.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

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, 2026a. URL https://arxiv.org/abs/2510.04618.

Qizheng Zhang, Michael Wornow, Gerry Wan, and Kunle Olukotun. Agentic plan caching: Test-time memory for fast and cost-efficient llm agents, 2026b. URL https://arxiv.org/abs/2506.14852.

Yudi Zhang, Meng Fang, Zhenfang Chen, and Mykola Pechenizkiy. Self-evolving llm agents with in-distribution optimization, 2026c. URL https://arxiv.org/abs/2606.07367.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners, 2024. URL https://arxiv.org/abs/2308.10144.

Andrew Zhao, Yiran Wu, Yang Yue, 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, 2025. URL https://arxiv.org/abs/2505.03335.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. Skillweaver: Web agents can self-improve by discovering and honing skills, 2025. URL https://arxiv.org/abs/2504.07079.

Junhao Zheng, Chengming Shi, Xidi Cai, Qiuke Li, Duzhen Zhang, Chenxing Li, Dong Yu, and Qianli Ma. Lifelong learning of large language model based agents: A roadmap, 2026. URL https://arxiv.org/abs/2501.07278.

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, 2025. URL https://arxiv.org/abs/2508.16153.

A Profiling Experiments

This appendix reports the full protocol and the per-axis analysis behind the profiling study previewed in §1. The study isolates the two experience forms under the same task stream, and characterizes each along the three axes of Figure 1: *construction cost*, *execution efficiency*, and *transfer robustness*. The goal is to reveal the underlying features of each form and the trade-offs they bear to inspire how to design a more effective and efficient self-evolving agents.

A.1 Experimental Setup

Benchmark and roles. All profiling runs are on the AppWorld (Trivedi et al., 2024) `train` split (90 tasks). We fix a strong, widely-used executor, GPT-4o, and a separate, more capable reflector, Claude-Sonnet-4.6; keeping the two roles on different models isolates the cost of *constructing* knowledge (reflector) from the cost of *using* it (executor), and prevents a single model's idiosyncrasies from confounding both sides at once.

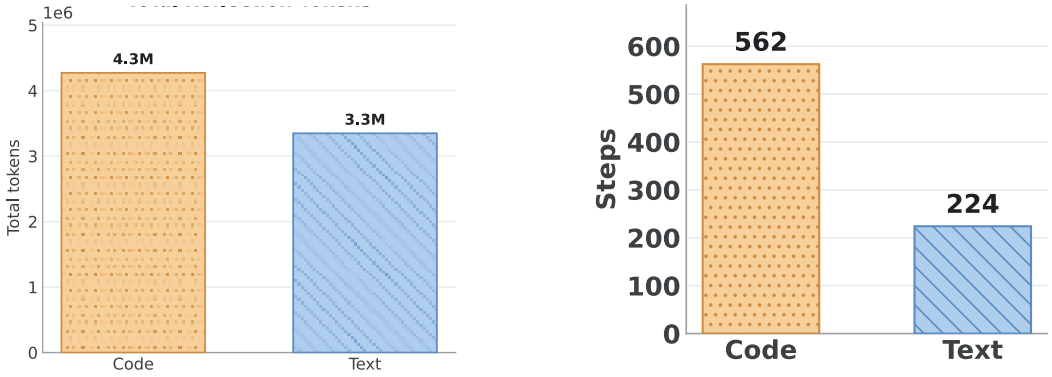
Execution Protocol. Unless stated otherwise, knowledge evolves in a streaming regime that mirrors deployment. Tasks are processed in a fixed order; after each task the reflector distills knowledge from that task's trajectory and commits it to memory, and the executor may retrieve and apply that memory on every subsequent task. Each task is therefore evaluated against a memory built only from strictly earlier tasks. To analyze the transfer robustness, we further conducted an extra experiment under oracle setting. Under this setting, we (i) first run all 90 tasks to collect their trajectories, (ii) reflect on each trajectory independently and in batch, so no online ordering effect or cross-task contamination is in play, (iii) assemble the full tool library, (iv) manually remove the single tool we identified as broken (*all other tools left intact, with no further edits*), and (v) re-run all 90 `train` tasks with that library injected. Two properties make this setting an upper bound rather than a generalization measure: the library is built from the entire task set and then hand-audited, and it is re-tested in-sample on the very tasks it was distilled from. We apply the same batch-reflection protocol to Text to obtain its Oracle point for comparison. We audited the text memory under the same criterion and found no entry that required removal, so both Oracle points represent a curated upper bound.

Conditions. We compare three conditions that share this protocol and differ only in the form in which experience is stored: **No Memory** (the knowledge-free baseline; the executor sees neither tips nor tools), **Text** (the reflector distills experience as natural-language text memories), and **Code** (after each task, the reflector reads that task's full trajectory, as in Text, and compiles its procedures into callable tools, without a recurrence gate). Both reflectors thus receive the same kind of input under the same task order, although their trajectories diverge once their memories differ. This condition differs from the Metis codifier, which never sees trajectories (Appendix A.3). A consequence of the two forms is an injection asymmetry that the analysis turns on: a text tip is injected directly into the executor's context, so the executor reads its full content and decides how to act on it; a code tool is compiled into a callable object, so the executor sees only its docstring and signature and invokes it as a black box, inheriting whatever its body does.

Metrics. We report task accuracy (task goal completion, TGC), the execution cost on the executor side (total tokens and ReAct steps), and the construction cost on the reflector side (total reflection tokens and LLM rounds). For execution efficiency we report both All Tasks and the Commonly Solved subset—the tasks solved by all three conditions. Commonly Solved is the apples-to-apples comparison: it removes the confound that the conditions succeed on different task sets, so a token or step difference reflects how each form solves the same work rather than which work each happened to finish.

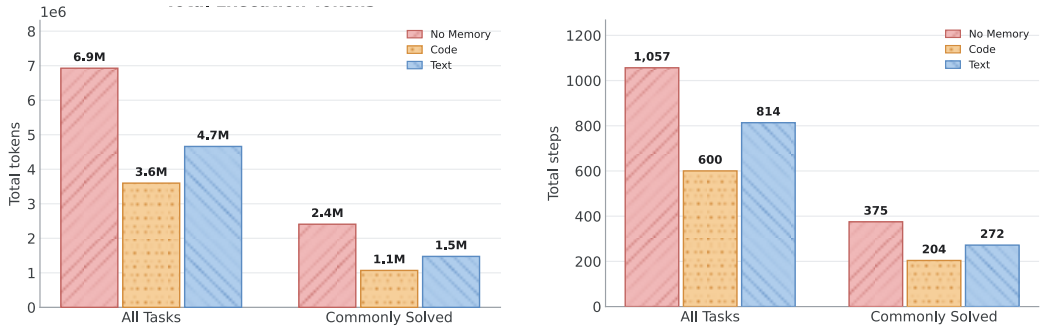
A.2 Per-Axis Analysis

Construction cost: text is cheaper to build. Over the streaming run, code reflection consumes 4.3M tokens across 562 LLM rounds, against text's 3.3M tokens across 224 rounds (Figure 5). The round count is the sharper signal: building code costs $2.5\times$ the LLM rounds of building text, while the token gap is a more modest $1.3\times$. The disparity follows from *how* each form is produced. Text is distilled in essentially a single read of the trajectory, so each round is one long prompt; code is produced interactively—the reflector drafts a candidate tool and must then inspect API documentation, execute the tool in a live sandbox, and debug it before the tool is admitted, which multiplies rounds while keeping each round short. We are deliberate about not overstating the token gap: per unit of memory, code is only moderately more expensive in raw tokens. The cost that matters is that code's heavier construction loop (sandbox execution, verification, retries) is, as the transfer robustness axis shows, *largely wasted under streaming*—the tools it pays to build do not transfer.



(a) Reflection tokens (b) Reflection rounds

Figure 5: Construction cost over the streaming run. Building code costs $1.3\times$ the reflection tokens but $2.5\times$ the LLM rounds of building text, because code must be tested and debugged in a live sandbox before a tool is admitted.

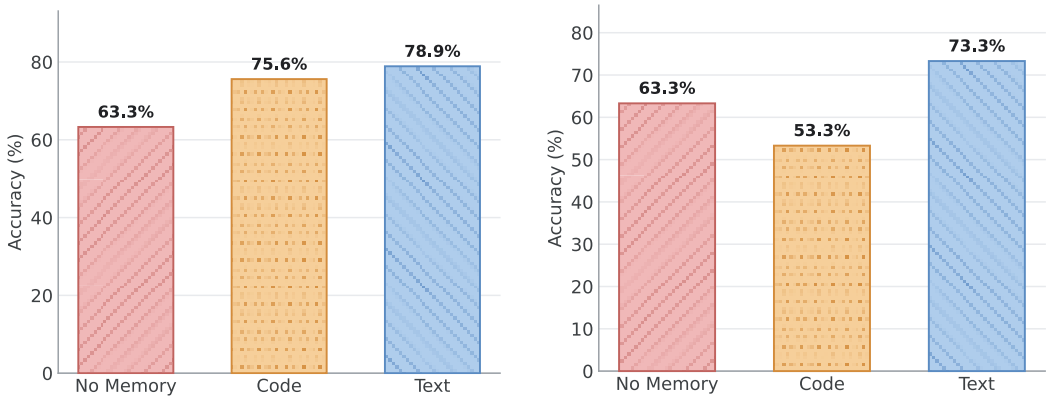


(a) Execution tokens (b) Execution steps

Figure 6: Execution efficiency. On the Commonly Solved subset (same tasks, all three conditions succeed), code cuts execution tokens and steps substantially more than text relative to the no-memory baseline.

Execution efficiency: code accelerates execution far more. On the Commonly Solved subset—the fair, same-tasks comparison—code drives the executor to 1.1M tokens and 204 steps, text to 1.5M tokens and 272 steps, against the no-memory baseline's 2.4M tokens and 375 steps (Figure 6). Both forms accelerate execution, but code's reduction is the larger by a wide margin: relative to baseline it cuts tokens by 54% and steps by 46%, where text cuts them by 38% and 27%—roughly 1.4–$1.7\times$ the per-task saving, and head-to-head code is about a quarter leaner than text. The mechanism is the injection asymmetry. A callable tool collapses an entire procedure—API discovery, composition, and boilerplate—into one invocation that *bypasses reasoning*, whereas a text tip must be read and reasoned over on *every* use: the agent still re-derives the procedure each time, only faster and more reliably. This is precisely why text yields little efficiency gain relative to code even though it does shorten executions versus the baseline.

Text memory has better transfer robustness. The third axis is the subtle one, and it has two facets. The first is *generalization*, read directly off the Oracle–Streaming contrast (Figure 7). Under the Oracle—an in-sample upper bound—code reaches 75.6%, $+12.3$ points over the 63.3% baseline and close to text's 78.9%. Under streaming forward transfer, code *collapses* to 53.3%, 10.0 points *below* the no-memory baseline, while text holds at 73.3%, still $+10.0$ above it. Code's Oracle$\rightarrow$Streaming drop is thus 22.3 points against text's 5.6. The reading is unambiguous: code's strong Oracle number is in-sample—the library is built from and tested on the same tasks and hand-cleaned—so it is an optimistic ceiling, and the gap between that ceiling and the streaming measurement is exactly code's



(a) Oracle (b) Streaming

Figure 7: transfer robustness. Under the in-sample Oracle, code nearly matches text; under streaming forward transfer, code collapses by 22.3 points to *below* the no-memory baseline, while text degrades by only 5.6 points and stays well above it.

failure to transfer to tasks unseen at build time. Per-task code reflection *overfits to the single trajectory it was distilled from*; text transfers with only minor degradation.

The second facet, which explains the first, is *safety*, and it again turns on the injection asymmetry. Reflecting over one trajectory, the reflector tends to crystallize tools that are over-specialized to that task—and sometimes generalized from an execution path that was itself *wrong*. Text reflection is prone to the same over-fitting, but the consequence differs sharply because of how each form is consumed. A text tip is in-context advice: the executor reads it and applies it with discretion, adapting it, partially applying it, or ignoring it when it does not fit, and it continues to sample the environment and self-correct. A code tool is a black box that the executor calls verbatim, so a tool distilled from a faulty trajectory propagates its defect to every caller; we further observe that the mere presence of a confident-looking tool *suppresses* the agent's own recovery behavior—it trusts the tool's (possibly empty or wrong) return rather than re-checking the environment, a recovery loop that text-only runs routinely exercise. Text is therefore the safer carrier of freshly-distilled, not-yet-verified experience; code becomes safe only once a pattern has been verified—which is exactly the privilege the Oracle condition grants and that streaming code never earns.

From profiling to Metis. The two designs at the core of Metis are direct responses to the two failure modes this study exposes. The transfer robustness analysis shows that text is the right default carrier for cold-start, judgment-bearing, and not-yet-verified experience—but only if the reflector knows *what kind* of experience it is distilling and phrases it for safe, adaptive reuse. This is why Metis does not treat text as a flat playbook but organizes it by experience type, a taxonomy (environment facts, execution plans, execution pitfalls) we arrived at precisely while diagnosing which text distillations transferred and which misled—yielding *Differentiated Text Memory* (§3.2). The efficiency and cost axes show that code pays off handsomely once a pattern has genuinely recurred and been verified, yet firing code reflection per task is what produced both the streaming overfitting and the wasted construction cost. Metis therefore gates codification on the text memory and on observed recurrence, and validates each tool before admission—*Recurrence-guided Code Generation* (§3.3)—so the high construction cost of code is paid only for patterns that have already proven worth crystallizing.

A.3 Profiling Code Condition vs. the Code-only Ablation

The Code condition in this study and the Code-only variant in § 4.3 both codify experience after every task, but they differ in what the codifier observes. The profiling Code reflector reads the full trajectory of each task, as the Text reflector does, so that the two forms are compared at equal information density. Code-only instead reuses the Metis harness and codifier (§ 3.3, Appendix C), which receives task queries rather than trajectories and admits a tool only after dependency and compilation checks. As a result, Code-only's construction cost (4.89M) is on par with Text-only (4.96M) because its

input is far shorter than the trajectories read by the text reflector. That comparison is therefore not at equal information and does not revise the cost ordering in § A.2. The two settings also differ in protocol: streaming on the training split for profiling, and train-then-test on the dev split for the ablation. Under these differences, Code-only improves over No Memory by 4.1 points, whereas the trajectory-fed Code condition falls 10.0 points below it. This is consistent with our diagnosis that excluding trajectories and gating admission curb the overfitting of per-task code memory. Code-only nonetheless trails Text-only by 2.1 points, preserving the ordering between the two forms.

B Main Experiment Settings

B.1 Evaluation Protocol

All methods follow the same train-then-test protocol. For each benchmark, methods process the training tasks in the same fixed order to construct their memory, after which the resulting memory is frozen and evaluated on the test tasks. The same benchmark split, task order, and execution setting are used across methods. Some methods allow the training set to be revisited for multiple rounds of memory refinement (Wang et al., 2026; Zhang et al., 2026a). Since additional passes provide further interaction and update opportunities and can in principle benefit any self-evolving method, we restrict all methods to a single pass over the training tasks for a controlled comparison. This controls the amount of task exposure rather than forcing the methods to share the same update rule. In particular, we preserve each method's original criterion for when memory is constructed or updated. Methods such as ASI and SkillX that learn only from successfully completed tasks retain this behavior, while methods that reflect on every rollout continue to do so.

B.2 Method Configurations

We retain each method's original memory representation, construction procedure, and retrieval mechanism wherever applicable, and make only the adaptations required to operate in the corresponding benchmark environment. Benchmark-provided execution components are kept unchanged across methods. Below, we describe the configuration for each method.

Metis. We set the codification threshold θ to 3. For text-memory retrieval, we retrieve up to 10 entries from each of the three memory types, yielding at most 30 textual candidates in total. We additionally retrieve up to 10 code tools. These candidates are then passed to the Memory Manager for final selection.

ACE. We adopt ACE's official implementation with its default settings. ACE introduces no additional hyperparameters that require tuning in our setting. We therefore only make minor prompt adaptations to ensure that its context updates are compatible with each benchmark environment.

ASI. We adopt ASI's official implementation with its default settings. ASI introduces no additional hyperparameters that require tuning in our setting. We therefore only make minor prompt adaptations to ensure that it generates valid code tools for each benchmark environment.

SkillX. We adopt the official implementation of SkillX in its hybrid mode, which combines planning, functional, and atomic skills. Consistent with our single-pass protocol, as stated in Appendix B.1, we perform 1 rollout per training task and a single extraction round, and disable exploratory skills expansion, which introduces environment interaction beyond the training tasks. Retrieval follows the official defaults, and at most 10 skills are retained after filtering according to the official code default, and the minimum retrieval similarity is 0.45.

C Metis Harness

The Metis harness mediates every interaction between the memory library and the agents that read or write it. It serves three stages: (1) during reflection, it gives the reflector controlled access to execution evidence; (2) during codification, it admits a generated tool into $\mathcal{M}_{\text{code}}$ only after dependency and

compilation checks; and (3) during execution, it assembles a dependency-complete tool context for the executor. Stages (2) and (3) share a single dependency-closure operator, which we define first.

Dependency Closure. Retrieval and the Memory Manager select memory entries individually and are unaware of dependencies between them. A selected plan may call a library tool that was not selected, and a selected tool may call helper functions that were not selected. Without these dependencies, the executor would be instructed to call functions that are absent from its context. The harness therefore completes the selection deterministically through static dependency closure, so that the Memory Manager only judges relevance and never needs to reason about implementation details.

Formally, we write $\mathrm{Ref}(m) \subseteq \mathcal{M}_{\mathrm{code}}$ for the library tools referenced by a text memory m (for an execution plan p, this includes its implementation set $D(p)$), and $\mathrm{Dep}(t) \subseteq \mathcal{M}_{\mathrm{code}}$ for the custom library functions called by the implementation of tool t. Given selected text memories S^{text} and tools S^{tool}, the harness starts from

$$U_0 = S^{\mathrm{tool}} \cup \bigcup_{m \in S^{\mathrm{text}}} \mathrm{Ref}(m),$$

and repeatedly adds the dependencies of every tool in the set,

$$U_{r+1} = U_r \cup \bigcup_{t \in U_r} \mathrm{Dep}(t),$$

until no new tool is added ($U_{R+1} = U_R$), which handles helpers that call other helpers. We denote the result by $\mathrm{Close}(S^{\mathrm{text}}, S^{\mathrm{tool}}) = U_R$.

Progressive Disclosure for Reflection. Raw trajectories can be long and noisy, so the harness does not place the full trajectory into the reflector context. Given a trajectory $\tau_i = \big(q_i, (a_1, o_1), \ldots, (a_T, o_T)\big)$ as defined in Section 2, the harness compresses each step independently into a compact view $\tilde{\tau}_i = (\tilde{s}_1, \ldots, \tilde{s}_T)$ with

$$\tilde{s}_t = \big(\mathrm{type}(a_t),\ \mathrm{name}(a_t),\ \mathrm{Trunc}(a_t; h),\ \mathrm{Trunc}(o_t; h)\big),$$

where $\mathrm{type}(a_t)$ is the action type, $\mathrm{name}(a_t)$ is the tool name when applicable, and long fields keep only their head and tail spans:

$$\mathrm{Trunc}(x; h) = \mathrm{head}_h(x) \,\|\, \ldots \,\|\, \mathrm{tail}_h(x).$$

This view gives the reflector a global picture of the execution without repeatedly injecting long observations, intermediate states, and tool outputs. The reflector can then expand details on demand through inspection tools: it may request the full content (a_t, o_t) of a specific step, inspect a bounded range of steps, or view the implementation of an existing library tool when code-level reasoning is required. Reflection therefore proceeds from a lightweight global sketch to targeted local inspection. This access is given to the text reflector only, the codifier receives no trajectory.

Tool Codification and Admission. The codifier is instructed to synthesize small composable helpers rather than end-to-end solvers tailored to individual queries. Each helper must be supported by at least two structurally distinct queries from $Q_B(p) \cup Q_R(p)$, and task-varying elements such as entities, time ranges, filters, and API options must be exposed as parameters rather than hard-coded. For a candidate tool t, the harness identifies its calls to existing library tools and helper functions, and validates it in the same sandbox environment used by the executor with all dependencies injected through the closure:

$$\mathrm{Compile}\left(\mathrm{impl}(t), \mathrm{Close}(\emptyset, \{t\})\right) = 1.$$

If compilation fails, the codifier receives the error message and is allowed one corrective round. A second failure rejects the candidate. Admitted tools are added to $\mathcal{M}_{\mathrm{code}}$ and registered in $D(p)$, and the consumed candidate buffer is folded into the related-task history $R(p)$. Subsequent text reflection may then revise p to reference the new tools. Recurring plans thus give rise to tools, and admitted tools in turn shape future plans, closing the loop between text memory and code memory. If no candidate is admitted, $B(p)$ is kept, so codification is retried the next time p is selected.

Tool Injection at Execution. For task i, after the Memory Manager selects text memories S_i^{text} and tools S_i^{tool}, the harness injects the executable tool context

$$U_i^{\star} = \mathrm{Close}\left(S_i^{\mathrm{text}}, S_i^{\mathrm{tool}}\right).$$

This context contains the tools directly selected for the task, the tools referenced by the selected text memories, and all custom helper functions required by those tools. Tool injection is therefore dependency-complete without requiring the Memory Manager to reason over implementation details.

D BENCHMARK DETAILS

AppWorld. AppWorld (Trivedi et al., 2024) is a stateful interactive benchmark built around 9 simulated everyday applications exposing 457 APIs. Its environment is populated with realistic digital states for fictitious users, such as emails, purchases, notes, and financial records. The benchmark contains tasks that require agents to inspect the current environment, compose multiple API calls, and often coordinate actions across applications. Tasks are evaluated using programmatic state-based tests, which verify not only whether the requested goal is achieved but also whether the agent introduces unintended changes to the environment. AppWorld therefore emphasizes long-horizon tool use, cross-application reasoning, and accurate manipulation of persistent environment state.

OfficeBench. OfficeBench (Wang et al., 2024c) evaluates agents on realistic office automation workflows involving applications such as Word, Excel, PDF, Calendar, Email, and Shell. The benchmark contains 300 tasks ranging from single-application operations to workflows that require coordinated actions across multiple applications. Completing a task may require the agent to locate and manipulate files, extract or transform information, switch between applications, and propagate intermediate results across a sequence of dependent operations. OfficeBench uses task-specific evaluators, including exact matching, fuzzy matching, and execution-based checks, to determine whether the required workflow has been completed correctly.

STATE-Bench. STATE-Bench (Microsoft, 2026) is a benchmark designed around stateful enterprise workflows. It contains 450 tasks across three domains: Travel, Customer Support, and Shopping Assistant, with 100 training tasks and 50 evaluation tasks per domain. Each task instantiates a pre-populated environment containing domain-specific entities such as bookings, orders, and customer records, and requires the agent to interact with both the environment and an LLM-based user simulator through multi-turn conversations. Successful execution requires more than reaching the correct final state. The agent must also follow domain-specific procedures, gather missing information from the user, present relevant options or consequences, obtain authorization when required, and invoke tools in the appropriate order. Task success therefore considers both environment-state correctness and interaction quality. An LLM judge evaluates whether the agent's communication follows the required protocol, and a task is considered successful only when both the task goal is accomplished and the interaction satisfies the protocol requirements. STATE-Bench thus stresses the ability to retain and reuse procedural knowledge about both environment operations and user-facing interaction constraints. For all methods, we use DeepSeek-V4-Flash as both the simulated user and the LLM judge to keep the evaluation environment consistent across methods.

E CASE STUDIES

We illustrate how each memory type in Metis turns a failed execution into a successful one. For each type, we select a task on which the Base Agent fails and Metis succeeds with the same executor model (Figures 8–11). Each figure shows the task, the cause of the Base Agent's failure, the memory entry that Metis selects, and how Metis completes the task. For readability, we lightly edit the memory entries and trajectories while preserving their content.

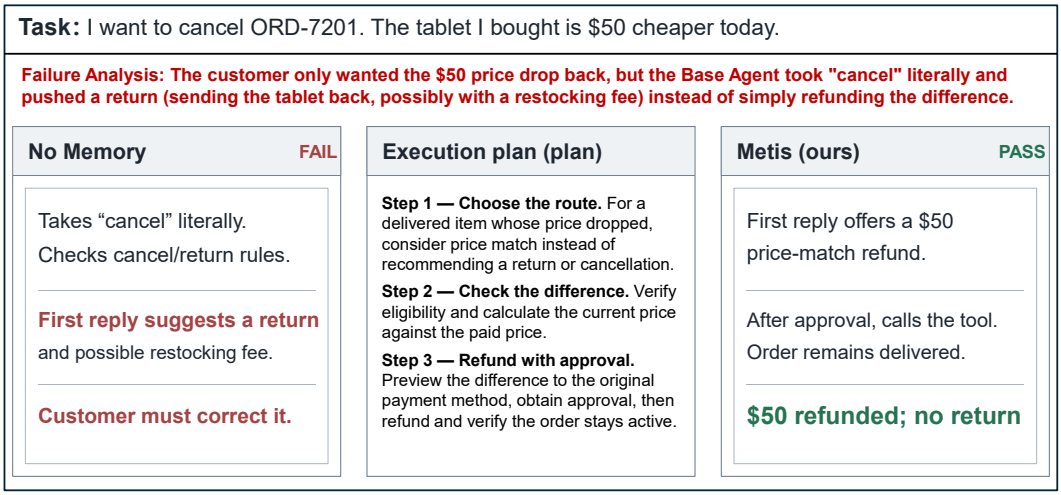


Figure 8: Case study of execution plan memory.

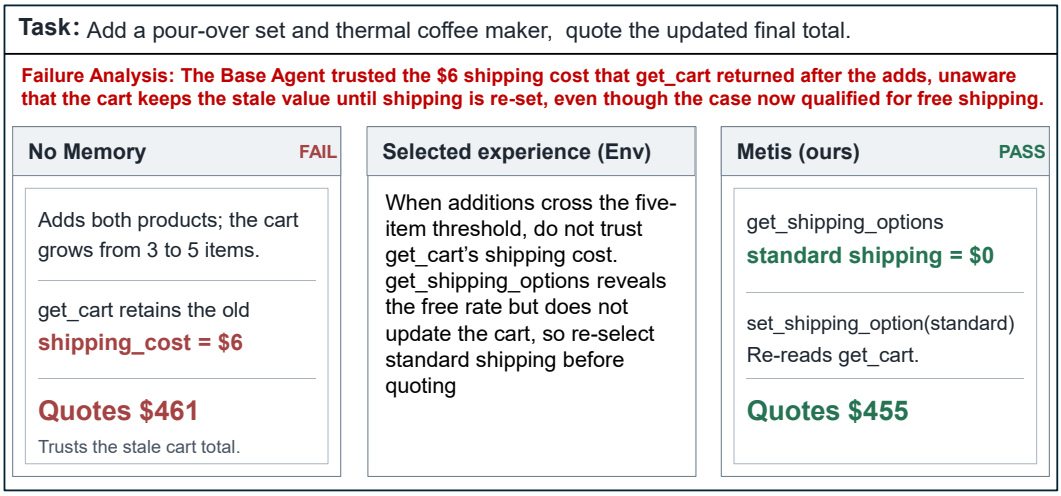


Figure 9: Case study of environment fact memory.

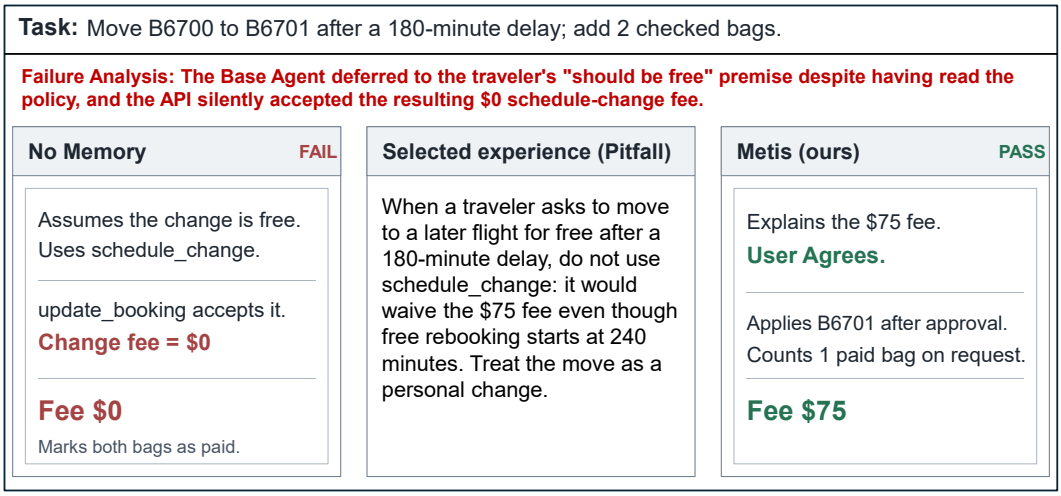


Figure 10: Case study of common pitfall memory.

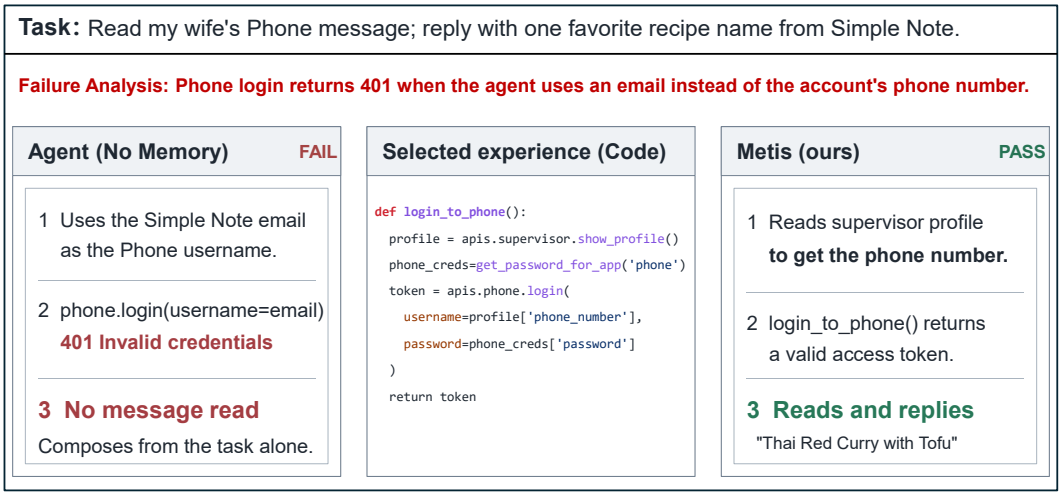


Figure 11: Case study of code memory.

F PROMPTS

This appendix presents the example prompt templates for the three components of Metis, in the order introduced in Section 3.

F.1 TEXT MEMORY REFLECTOR

The Text Memory Reflector distills execution trajectories into reusable environment facts, execution plans, and common pitfalls, and maintains existing text memories.

F.1.1 TASK AND EVIDENCE REQUIREMENTS

Text Memory Reflector – Part 1/6

SYSTEM:
You are a knowledge extraction agent.
USER:
You are analyzing an agent's execution trajectory on a task to extract reusable knowledge entries that will be injected into the agent's prompt for future tasks.
Distill **0-3 high-quality reusable entries** from each trajectory. Return an empty list `[]` when no candidate passes the evidence, novelty, and semantic quality bars below. Never emit filler to reach a target count.

Executor Environment

<environment-specific execution model and runtime constraints>

Evidence and Novelty Policy

1. Use this evidence order: executed code, observed output, explicit errors, and results obtained through Interactive Exploration are primary evidence. Thought and Reasoning are untrusted intentions. A binary ground-truth label, when present, is only a coarse evaluator signal. Never let intention or a coarse outcome override what the executed trajectory records.
2. `Fail` means only that some postcondition was not satisfied. It does not identify the cause. Attribute a cause only when the trajectory contains an explicit error, a contradictory read-back, or another observed action/state mismatch. If the evidence supports only a hypothesis, return `[]`.
3. ENVIRONMENT requires directly observed stable behavior. Do not turn a guess, proposed workaround, or task-specific observation into a universal fact. EXECUTION_PITFALL requires a complete observed chain: trigger -> actual mistake -> visible error or wrong state -> consequence. If any link is missing, do not emit the pitfall.
4. Create or update only when the reusable contract gains a parameter surface, invariant, postcondition, boundary, or counterexample. A new filename, person, date, record value, task wording, or example is not new knowledge. Do not append examples to an already adequate entry. Keep each entry's content at no more than 120 English words; provenance belongs in `source_task_ids`, not in the content.

Figure 12: Text Memory Reflector prompt: task and evidence requirements.

F.1.2 EXPLORATION TOOLS AND MEMORY TYPES

Text Memory Reflector – Part 2/6

Interactive Exploration

You have access to the same API documentation surface as the executor and can execute arbitrary Python code in a sandbox. **Use it** to understand the APIs involved in this trajectory and to verify hypotheses before writing entries:
<API discovery and documentation tools, distilled-tool inspection, and execution-trajectory inspection interfaces and calling conventions>
This is an interactive session. Write ONE ```` ```python ```` code block at a time, then STOP and wait for the execution result before continuing. Do NOT write multiple code blocks or a ```` ```json ```` block in the same response as a code block. When you are done exploring and ready to output entries, write a ```` ```json ```` block containing a JSON array (with NO code blocks). This ends the session.
Example interaction flow:
<example interaction flow>

Knowledge Entry Categories

Each entry belongs to exactly one category. Every entry must begin with a "When [condition]..." applicability clause.

1. **ENVIRONMENT**: A directly observed, stable fact about the sandbox, an API's behavior, a data model, or a runtime constraint that holds regardless of the particular task.
   GOOD: <example>
   BAD: <example>
2. **EXECUTION_PLAN**: A reusable parameterized semantic sub-workflow. It must identify a concrete domain object, a fixed domain operation over resolved inputs, and an observable postcondition. It must not be a whole-task script, an implementation transcript, or termination guidance. Do not mention <environment-specific task-completion interfaces> or their arguments.
   Dependent tools. If your plan's procedure would invoke any of the distilled code tools listed below in the `Existing Distilled Code Tools` section, add their function names to the plan's optional `dependent_tools` field (exact names, list of strings). The downstream Manager uses this list to ensure that when the plan is selected for a future task, the tools it relies on are surfaced alongside it — without this, plan and tool selection drift apart and the agent re-implements the procedure inline. Only list a tool if your plan's steps would actually call it; do not enumerate every tool in the same app. Omit the field entirely when no existing tool applies.
3. **EXECUTION_PITFALL**: A generalizable warning backed by an observed failure or inefficiency chain. It must name the trigger condition, the actual mistake, and the visible consequence. The pattern must apply to a class of tasks, not just the observed instance.
   GOOD: <example>
   BAD: <example>

Figure 13: Text Memory Reflector prompt: exploration tools and memory types.

F.1.3 Defining and Updating Execution Plans

Text Memory Reflector – Part 3/6

EXECUTION_PLAN Identity and Update-First Policy

1. Define every plan by **Object + Operation + Postcondition**:
   - Object: the domain object being changed or inspected, such as a playlist, email thread, calendar event, note, payment, or shopping cart.
   - Operation: the domain action applied to that object, such as add resolved songs, aggregate matching records, reschedule an event, or reconcile two collections.
   - Postcondition: the observable state that must hold after the operation, such as a playlist containing exactly the resolved songs or an event preserving the requested duration at its new time.

   Python, loops, API-document lookup, and <environment-specific API calls> are implementation choices. They must not define the plan identity. GOOD: <example>
   BAD: <example>
2. Write EXECUTION_PLAN content as a semantic contract, not an execution transcript. Use this shape:
   `When <trigger>. Object: <domain object>. Operation: <fixed domain action over resolved inputs>. Postcondition: <observable resulting state>. Boundary: <dynamic discovery or judgment left to the caller, if relevant>.`
   Object, Operation, and Postcondition are required. Boundary is optional. Parameters such as IDs, names, dates, filters, quantities, directions, and destination collections may be resolved inputs. A Boundary may leave dynamic target discovery or final judgment to the caller, but it must not hand back a core computation the Operation claims to perform. If the caller already computed every output value, describe the plan as the remaining read/write/mutate kernel. Do not include Python source, exploratory API-doc calls, retry transcripts, or benchmark completion instructions.
3. Route implementation knowledge to the correct category:
   - ENVIRONMENT records stable facts about an API, runtime, data model, or sandbox behavior.
   - EXECUTION_PITFALL records a trigger, implementation mistake, and observed consequence.
   - Incidental implementation details that are neither stable facts nor reusable pitfalls should not become knowledge.
4. Before creating a plan, compare Object + Operation + Postcondition against EVERY existing EXECUTION_PLAN in retrieved knowledge and the active library. Compare meanings, not exact IDs or wording. Treat different people, dates, filenames, record values, filter values, ordering directions, or API-call spellings as parameters of the same plan when the reusable operation and postcondition remain the same.
5. UPDATE-FIRST is mandatory, but an update also needs semantic novelty. If an existing plan has the same object, operation, and compatible postcondition and parameter surface, keep it unchanged unless this trajectory adds a real parameter, invariant, postcondition, boundary, or counterexample. Preserve the stable ID when updating the same concept and replace the contract with one compact generalized description. Create a new plan only when no active plan represents the same reusable capability.
6. Do not over-merge merely because tasks use the same app or implementation. Searching emails, aggregating senders, and mutating labels are different operations. Conversely, different search terms or sort directions are normally parameters of one search or sort plan.
7. A read-only index entry still counts as an existing plan. If it is a semantic match but is not an allowed update target, do not create a duplicate; emit no plan entry for that concept.

Few-shot examples:
GOOD: <example>
BAD: <example>
<existing tips section>
<existing tools section>

Figure 14: Text Memory Reflector prompt: defining and updating execution plans.

F.1.4 Reviewing Trajectories and Proposing Memories

Text Memory Reflector – Part 4/6

Recent Task Queries

Task instructions from recently-reflected tasks (FIFO buffer, oldest first). Use these to spot recurring reasoning moments — a sub-routine that fits multiple of these queries is a stronger EXECUTION_PLAN candidate than one fitting only the current task. These are NOT ground truth — do not assume the patterns suggested here generalize without evidence in the current trajectory.

- <recent task query>

<retrieved section>
<recently codified hint>

Trajectory

Long `[Output]` step blocks are rendered as `<head> [... TRUNCATED BY HARNESS: N of M chars hidden (head=..., tail=..) ...] <tail>`. That marker is harness-side rendering only — NOT evidence of API pagination, preview ellipsis, or any data-side truncation. Do not infer pagination bugs or "missing data" entries from its presence. If the hidden middle would actually affect your decision, call `show_trajectory_steps(task_id='<task id>', start=N, end=N+1, truncate=False)` to read step N's full output before deciding (recall: 0-indexed, end-exclusive).
Task ID: <task id> Task: <task query> Steps total: <n steps><ground truth section>
Steps: <formatted steps>

Analysis Steps

Before outputting entries, walk through this analysis in your response:

1. **TRAJECTORY READING**: Reconstruct what actually happened from executed code, outputs, and errors. Check every task constraint and subgoal. Treat Thought/Reasoning as claims to verify, and never infer a cause from a coarse Success/Fail label alone.
2. **CANDIDATE PROPOSAL**: Propose at most three distinct reusable candidates. For every candidate, name its concrete evidence and the behavior it changes in a different but similar task. Reject unsupported hypotheses, instance details, generic advice, and implementation trivia. For each plan, explicitly identify its Object, Operation, Postcondition, and optional Boundary before drafting it.

Figure 15: Text Memory Reflector prompt: reviewing trajectories and proposing memories.

F.1.5 COMPARING AND FILTERING MEMORY CANDIDATES

Text Memory Reflector – Part 5/6

3. **UPDATE-FIRST SEMANTIC COMPARISON**: Before every `create`, compare the candidate's reusable meaning against every active entry shown in full. For a plan, compare Object + Operation + Postcondition, not wording, IDs, instance values, app names, or implementation details. If the same contract already exists and needs no substantive improvement, emit nothing for that concept.

Use `action: "update"` only when this trajectory adds a concrete parameter surface, invariant, postcondition, boundary, or counterexample. Preserve the existing stable ID when the reusable concept remains the same. The old target entry is invalidated and replaced by one compact generalized contract.

Primary update targets are entries in the `## Knowledge Retrieved for This Task` section — those are what the executor actually saw, so this trajectory is direct evidence for their quality. Entries from the `## Knowledge Library` section (not retrieved) are still updatable but you have weaker evidence about their effect; update them only on clearly transferable, directly observed evidence.

Merge only same-category entries that describe one reusable capability and can be replaced without losing distinct operations or postconditions. Sharing an app or implementation is not enough.

4. **QUALITY GATES** (Delete or adjust every candidate that fails any gate):
   - **Generalization**: ENV → fact holds regardless of task; PLAN → same template fits a sub-goal inside different task contexts; PITFALL → can name different distinct scenarios where the trap could be hit.
   - **Semantic novelty**: compare meanings against every existing entry and against the other candidates. If the same reusable contract already exists, update it only when evidence adds a real invariant, parameter, postcondition, boundary, or counterexample; otherwise DISCARD.
   - **Mechanism**: trigger condition is concrete (no "be careful with X"), mechanism is named (no "pagination can be tricky"), evidence is anchored to the trajectory (not generic meta-advice).
   - **Correct Scope** Be careful with the usage scope and names of your entries. Check the API doc for possible scenario misunderstanding and do not overstate the scope. e.g.: A plan guides to rate songs from playlist should not claim for album or library scope. When you detect such potential confusion, make sure to mention it to warn the agent.
   - **Behavioral Value**: the entry must change a future agent's behavior. If the future agent would act the same way without this entry, DISCARD it.
   - **No Termination Policy**: DISCARD or rewrite any candidate that tells the future agent to invoke <environment-specific task-completion interfaces>, return a count, return a summary, or pass/omit <environment-specific completion arguments>. Completion mechanics are executor-level instructions, not reusable plan/pitfall/environment knowledge.
5. **YIELD**: Return 0-3 accepted entries. Prefer `[]` over weak extraction.

Figure 16: Text Memory Reflector prompt: comparing and filtering memory candidates.

F.1.6 MEMORY OUTPUT FORMAT

Text Memory Reflector – Part 6/6

Output Format

Output as ```` ```json ```` array. Return `[]` if no candidate passes all gates.

Every entry MUST include an `action` discriminator: `"create"` to emit a brand-new knowledge entry, `"update"` to invalidate one or more existing entries of one category and replace them with a generalized entry. Missing / malformed `action` causes the entry to be rejected.

```
[
  {
    "action": "create",
    "id": "env__... | plan__... | pitfall__...",
    "label": "ENVIRONMENT | EXECUTION_PLAN | EXECUTION_PITFALL",
    "content": "When [condition]... <actionable advice, 1-4 sentences>",
    "source": "success | failure | inefficiency",
    "dependent_tools": ["existing_tool_function_name", ...]
  },
  {
    "action": "update",
    "target_ids": ["existing_tip_id_1", "existing_tip_id_2"],
    "id": "stable_semantic_id",
    "content": "<new merged tip body>",
    "source": "success | failure | inefficiency",
    "dependent_tools": ["existing_tool_function_name", ...]
  }
]
```

`dependent_tools` is OPTIONAL, EXECUTION_PLAN only — omit when no existing distilled code tool applies. Ignored on ENV / PITFALL updates.

For `update`: `target_ids` is a list of one or more EXISTING tip IDs that will be invalidated. The new tip's category is INFERRED from the targets — all target IDs must resolve to tips of the same category. Keep the existing stable ID when updating one concept. IDs identify reusable semantics, not provenance: use lowercase snake_case with exactly one `env__`, `plan__`, or `pitfall__` prefix, and never include task IDs, people, dates, filenames, record values, or other instance details.

Figure 17: Text Memory Reflector prompt: memory output format.

F.2 CODE MEMORY REFLECTOR

The Code Memory Reflector turns recurring execution plans and their associated task queries into reusable Python helper tools, with sandbox validation.

F.2.1 TASK AND SANDBOX TOOLS

Code Memory Reflector – Part 1/4

SYSTEM:
You are a knowledge extraction agent.
USER:

Task

You distill reusable Python helper tools that accelerate an agent's future execution. You will be shown:

- A **Plan** that has already been reused across multiple distinct tasks. It's indicating a shared pattern among the referenced tasks.
- The task queries (candidate + related) that applied this plan.
- The existing distilled tool library.
- A live Python sandbox with the same API surface the executor uses.

Your job: identify high-reuse routines that are *shared across the queries* and worth being callable functions. The plan is a **cluster anchor** — it tells you these queries belong to one class. Returning `[]` is fully acceptable and often correct if you think current tool library is already sufficient.

Executor Environment

<environment-specific execution model and runtime constraints>

Sandbox Configuration

Testing Environment

The sandbox you test in is loaded with the environment of ONE specific candidate task, with <environment-specific fixture configuration>. The OTHER candidate / related queries belong to **different task environments**; their named entities do not exist in this sandbox. When you test a tool implementation, use values you discover from THIS sandbox's actual data — <environment-specific data-discovery procedure>; do not lift names from the other candidate queries. If your tool depends on data this env does not have, that does not invalidate the tool — it just means you cannot verify it here.
This grounding is provided so you can run real tests. **It is NOT a signal about codification priority.** Codify for the queries' collective scope; the specific task we put you in is incidental.
Sandbox context:
<sandbox context>
This sandbox's task query:
<testing task query>

Sandbox Tools

You have the following tools available: <environment-specific API discovery and documentation tools>

- `show_tool_doc('function_name')` — print the full doc (docstring, parameters, implementation) of an existing **distilled code** tool. Environment's API docs are NOT accessible via this — use <environment-specific API documentation tools> instead. This tool prints internally; call it bare, do NOT wrap in `print(...)`.
- Existing distilled tools are live callables: `result = tool_name(apis, ...)`.

Interaction protocol

- Exactly ONE ```` ```python ```` block per turn, then STOP and wait for the result.
- The final ```` ```json ```` array ends the session. We take the LAST ```` ```json ```` block in your output — do not emit intermediate drafts.

Figure 18: Code Memory Reflector prompt: task and sandbox tools.

F.2.2 CREATING AND VALIDATING REUSABLE TOOLS

Code Memory Reflector – Part 2/4

Quality bar

A tool earns its place only if it meets ALL of:

1. **Generality across the class.** Name at least two structurally different queries (candidate or related) that would call it. Tools that fit only one visible query are instance solutions, not tools.
2. **Fully parameterized.** Every dimension the API can vary is a parameter (auth tokens, IDs, filters, formats, search keys, page counts ...). Default posture is *more* parameters: silence in the queries is not evidence a dimension is fixed. In most cases hardcoding some values is a bad idea. If you decide to hardcode some entries, make sure you state it in the docstring.
3. **Real work, not a rename.** The body should add logic beyond a single API call: e.g.: credential lookup, multi-call composition, data transformation, error normalization, a domain calculation etc.. Pure pass-through forwarding is discarded.
4. **Verified in the sandbox.** Parameter names and types are checked via <environment-specific API documentation tools> (do not guess), and you have run the function with realistic inputs and seen it work.
5. **Stable semantic subworkflow.** A tool should complete one stable, independently nameable subgoal. Merge adjacent operations only when no agent judgment is needed between them, and never cover the full user request. Its boundary must fit one sentence: `BOUNDARY: expects <pre-resolved inputs>; owns <operations and postcondition>; leaves <discovery or judgment> to caller.` If you cannot state that boundary unambiguously, discard the candidate.

The examples below are illustrative shapes, not an exhaustive taxonomy:
GOOD: <example>
BAD: <example>

Working procedure

1. Read the plan and the queries. Name the potential tool helper(s) that can be reused across at least two queries. If you cannot name one, return `[]`.
2. Before creating any candidate, inspect the most semantically similar existing tool(s) with `show_tool_doc`. If one existing tool already achieves the same subgoal and postcondition, discard the candidate; do not create a renamed, narrower, or lightly extended duplicate. Existing tools that cover only internal steps do not block a stable semantic subworkflow.
3. Never guess. Look up the relevant APIs with <environment-specific API documentation tools> when needed. Decide the parameter surface — every API-variable dimension is a parameter unless you can justify fixing it from the API itself.
4. Implement and test in the sandbox with realistic inputs drawn from the queries. If the test fails and fixing it requires unverifiable assumptions, discard.
5. Emit the final ```` ```json ```` array.

Figure 19: Code Memory Reflector prompt: creating and validating reusable tools.

F.2.3 TOOL OUTPUT FORMAT AND DOCUMENTATION

Code Memory Reflector – Part 3/4

Output schema

Final output is a single ```json array. The following schema is provided for illustration,entries in parameters are just shown for illustration, not a requirement.

```
{
  "function_name": "verb_noun",
  "docstring": "<docstring you wrote>",
  "parameters": {
    "apis": {"type": "Any", "description": "The apis object"},
    "access_token": {"type": "str", "description": "App access token"},
    "query": {"type": "str", "description": "Optional keyword filter; null = no filter", "default": null}
  },
  "return_annotation": "str | int | list | dict | Any (parameterised forms like list[dict] are fine)",
  "implementation": "Function BODY only (no def line). Must use <environment-specific API interface>."
}
```

Schema rules — violations crash the tool factory:

1. Required parameters omit `default`. Optional parameters MUST include `default` with a concrete JSON value (`null`, `0`, `""`, `false`, etc.). Docstring and schema must agree: if the doc says "optional", the schema needs a `default`.
2. **NEVER use `**kwargs` in the implementation body.** The factory builds the signature from `parameters`, so `**kwargs` raises NameError. Forward optional filters as explicit params and assemble a local dict. `**params` is safe when `params` is a local dict, not the function's signature.

<example>

3. **Reflect on edge cases; raise on each — never substitute None/[] for an error.** Walk through what could degenerate the operation (empty result, missing key, unexpected enum, API failure payload, meaningless input). Raise an exception that names what was attempted. A `None`/`[]` return on a real failure looks like success to the agent, exceptions are the only signal it can't miss.

Docstring spec. The docstring is the agent's only surface when deciding to call. It must contain:

- One-line summary of what the tool does.
- WHEN clause: the trigger condition (which sub-goals / query shapes call this).
- One-sentence `BOUNDARY: expects <pre-resolved inputs>; owns <operations and postcondition>; leaves <discovery or judgment> to caller.`
- Returns (Also check if your claim matches your return value)
- Behavior on errors and edge cases (One line per failure mode)
- Per parameter: valid values, conventions, and what `null` / default means if optional.
- Example (ONE minimal usage; avoid showing every flag combination)

Do NOT include:

- More than one usage example unless variants are functionally distinct.
- Detailed Procedure. The agent can check the implementation if it wants. Do not overstate the function's scope; express its exact responsibility in the required `BOUNDARY` sentence instead of adding separate procedure detail.

Figure 20: Code Memory Reflector prompt: tool output format and documentation.

F.2.4 INTERACTION EXAMPLE, INPUT PLAN, AND TASK QUERIES

Code Memory Reflector – Part 4/4

Minimal interaction example

<example interaction flow>
Return `[]` if no candidate passes the quality bar.
<existing tools section>

The Plan

<plan content>

Candidate queries (triggered this codify pipeline)

<candidate queries section>

Related queries (same plan, broader scope)

<related queries section>
Reminder before you begin: the sandbox you operate in is loaded with the env of one specific candidate task (query: "<testing task query>"). Test against THIS env's real data; do not assume the other candidates' entities exist here.
Now begin.

Figure 21: Code Memory Reflector prompt: interaction example, input plan, and task queries.

F.3 MEMORY MANAGER

The Memory Manager selects relevant text memories and code tools that can support concrete substeps of the current task.

F.3.1 SELECTING RELEVANT MEMORIES AND TOOLS

Memory Manager – Part 1/1

USER:
You are a Task Orchestrator. Your mission is to analyze the task and select the exact set of text tips AND code tools that the execution agent will need — in ONE consistent decision.

Task Information
The task will be executed by a ReAct execution agent. Task description: <task query>

Available Text Tips (knowledge from prior experience)
Each tip has an ID, a category, and advice content.
<formatted tips>

Available Code Tools
Each tool has an ID, a function name, a description, and its parameter signature. (Implementations are omitted; rely on description + signature.)
<formatted code tools>

Selection Instructions
Select only the tips and tools that can support a concrete sub-step of the task.

- The item's scope, assumptions, and supported operations must match the task.
- Select applicable pitfall tips when they warn about an operation or tool relevant to the task.
- Exclude items that are unrelated, only superficially similar, or intended for a different scope.
- Select a code tool only when its documented capability is useful for the task.
- Dependencies of selected execution plans and code tools are added automatically after selection. Do not select an item solely to satisfy a dependency.
- It is valid to select nothing when no candidate is clearly useful.
- Completion policy: Do not select tips that prescribe <environment-specific task-completion interfaces> or answer-return mechanics. The executor owns completion.

Return format instructions:
Return ONLY ONE JSON object with EXACTLY two fields. Do not include reasoning, explanations, analysis, markdown fences, or any other text before or after it:

1. "selected_tip_ids": JSON array of relevant tip IDs (use [] if none).
2. "selected_tool_ids": JSON array of relevant tool IDs (use [] if none).

Example:

```
{"selected_tip_ids": ["tip_id_1", "tip_id_2"],
  "selected_tool_ids": ["tool_id_1"]}
```

To select nothing, return {"selected_tip_ids": [], "selected_tool_ids": []}. ID fidelity (MUST, case-sensitive): IDs must be EXACTLY as listed above. Do NOT invent or alter IDs.

Figure 22: Memory Manager prompt: selecting relevant memories and tools.