Memory Transfer Learning: How Memories are Transferred Across Domains in Coding Agents

Anonymous ACL submission

Abstract

Memory-based self-evolution has emerged as a promising paradigm for coding agents. However, existing approaches typically restrict memory utilization to homogeneous task domains, failing to leverage the shared infrastructural foundations, such as runtime environments and programming languages, that exist across diverse real-world coding problems. To address this limitation, we investigate **Memory Transfer Learning** (MTL) by harnessing a unified memory pool from heterogeneous domains. We evaluate performance across 6 coding benchmarks using four memory representations, ranging from concrete traces to abstract insights. Our experiments demonstrate that cross-domain memory improves average performance by 3.7% over zero-shot setting and even outperforms the strongest self-evolving baseline, primarily by transferring meta-knowledge, such as validation routines, rather than task-specific code. Importantly, we find that abstraction dictates transferability; high-level insights generalize well, whereas low-level traces often induce negative transfer due to excessive specificity. Furthermore, we show that transfer effectiveness scales with the size of the memory pool, and memory can be transferred even between different models. Our work establishes empirical design principles for expanding memory utilization beyond single-domain silos.

1 Introduction

As performance gains from scaling training data in language models begin to plateau, self-evolution, which leverages prior inference outcomes to enhance future performance without additional supervision, has emerged as a promising paradigm for advancing model capabilities in agents (Gao et al., 2025; Fang et al., 2025a). Memory plays a central role in self-evolving agents by enabling the extraction of reusable workflows and transferable insights from past inferences and their application

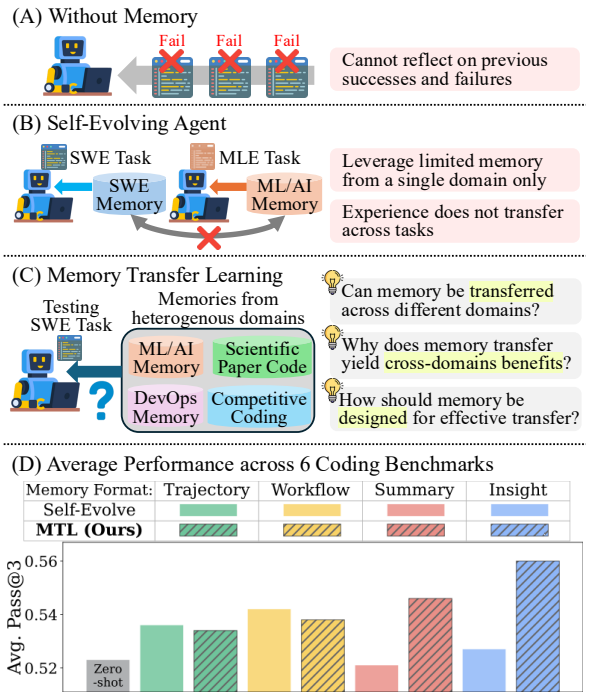

Figure 1: **Conceptual overview of Memory Transfer Learning.** Unlike (A) memory-less agents or (B) single-domain self-evolving agents, (C) our approach utilizes a shared memory pool from heterogeneous coding tasks. (D) In the evaluation on diverse coding benchmarks, MTL outperforms a self-evolving approach.

to subsequent tasks (Zheng et al., 2024; Wang et al., 2024c; Ouyang et al., 2025). In coding agents, the memory is instantiated as code snippets, experiential knowledge including planning and debugging traces, or general programming principles (Chen et al., 2025). Leveraging this knowledge allows agents to reference successful solution patterns in similar tasks, thereby reducing reasoning overhead, while also avoiding erroneous actions in code editing through adherence to accumulated procedural and strategic guidance, such as small-step modification heuristics and verification routines.

While memory-augmented coding agents have shown promise (Qian et al., 2024; Ouyang et al., 2025; Tang et al., 2025), existing approaches

mostly restrict memory generation and retrieval to the same domain, typically within the same benchmark as illustrated in Figure 1 (B). However, in real-world scenarios, coding agents must handle a wide spectrum of programming problems, ranging from repository-level software engineering tasks (Jimenez et al., 2024) and machine learning model development (Nathani et al., 2025; Seo et al., 2025) to function-level competitive coding (Jain et al., 2024). Despite this diversity, these tasks share a common underlying infrastructure, including runtime environments (e.g., Linux shells), programming languages, and cross-file dependency stacks. Current approaches that restrict memory utilization to a single domain fail to leverage this shared foundation, thereby preventing agents from exploiting a substantially richer memory pool derived from heterogeneous domains. We posit that such cross-domain memories can provide valuable guidance, often more effective than those extracted solely from the same domain, by offering transferable knowledge for new problems (Figure 1 (C)).

Some prior works have explored the construction of a large unified memory pool spanning multiple task types, providing initial evidence that general reasoning experiences can support software engineering tasks (Tang et al., 2025). However, this line of work leaves several key research questions for a practical deployment unresolved, as in Figure 1:

RQ1. *Does memory from heterogeneous domains improve the performance of coding agents?*

RQ2. *Why do transferred memories yield benefits across different coding domains?*

RQ3. *Which factors in memory transfer learning most affect transfer effectiveness?*

To address these open questions, we conduct a systematic investigation of **Memory Transfer Learning (MTL)** across heterogeneous domains in coding agents and derive several core findings on its mechanisms and effects. We first generate memories for each task using four different formats commonly adopted in prior works: Trajectory (Zheng et al., 2024; Zhao et al., 2024), Workflow (Wang et al., 2024c; Zheng et al., 2025), Summary (Shinn et al., 2023; Cao et al., 2025), and Insight (Ouyang et al., 2025; Suzgun et al., 2025), as illustrated in Figure 2. We then evaluate coding agent performance under three inference settings: zero-shot (without memory), self-evolving (with the same domain memory), and MTL (cross-domain memory). The results demonstrate that MTL can provide effective and transferable knowledge, **improving average scores by 3.7% of 6 coding benchmarks and outperforming self-evolving approaches.**

Our analysis yields three core findings on the mechanisms of memory transfer. **First, cross-domain memory significantly improves the performance of coding agents.** Although existing self-evolving methods often overlook out-of-domain memories, our results suggest that effective memory utilization should incorporate all past experiences, including those from different domains, to enhance agent performance, as shown in Figure 1 (D). **Second, the primary transferable value lies in meta-knowledge.** Through qualitative analysis, we find that cross-task benefits stem not from task-specific code content but from operational know-how, such as preventing execution failures under environment constraints and task-solving routines that prioritize structural and interface inspection followed by strict validation procedures. **Third, abstraction dictates transferability.** By quantifying the abstraction level of each memory format, we discover a positive correlation between high-level abstraction and transfer effectiveness. Highly abstract memories, such as Insight, become task-agnostic and generalizable. In contrast, low-abstraction memories like Trajectory retain excessive task-specific details that can distract the agent, confirming that raw execution traces are less suitable for cross-task transfer. Furthermore, we provide additional insights into MTL, including why negative transfer occurs, how the performance gain of MTL scales with a larger memory pool and more domains, and the potential for transferring memories across different models. We hope this study expands the scope of memory utilization beyond a single domain and stimulates further research on effective cross-domain memory strategies, ultimately leading to more capable coding agents.

2 Memory Transfer Learning

We introduce Memory Transfer Learning, which leverages memories generated from heterogeneous domains in coding environments. We verify the benefit of memory transfer by comparing the performance of the memory-augmented agent in three settings: zero-shot, self-evolving, and MTL.

2.1 Method

To investigate the impact of memory on the agent, we design a simple memory-based coding agent with a two-stage memory utilization process: mem-

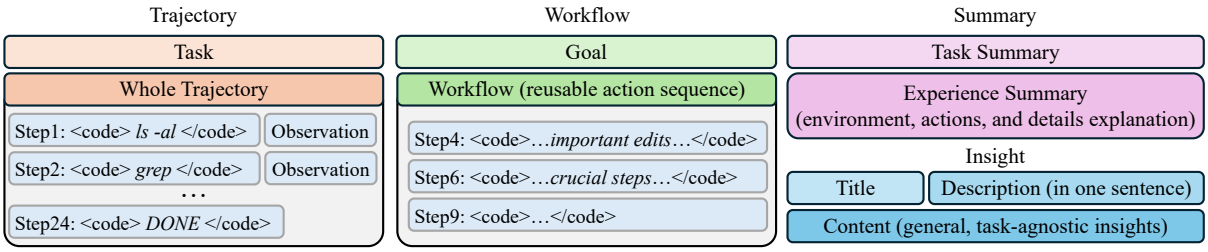

Figure 2: **Illustrative examples of four memory formats.** We utilize Trajectory, Workflow, Summary, and Insight formats to analyze how different levels of information abstraction affect cross-task transferability.

ory generation and memory retrieval. Memory generation is performed offline with results saved prior to memory transfer learning, while memory retrieval is executed for each query online.

2.1.1 Memory Generation

Before memory generation, we first run the agent across all benchmarks and gather the resulting trajectories as sources for memory construction. Inference results consist of the given task t and multiple steps of reasoning r, action a, observation o, thus the full inference history H is denoted as $H = (t, [(r_1, a_1, o_1), \ldots, (r_n, a_n, o_n)])$. Based on these results, we construct four types of memory representations, defined by categorizing memory schemes from existing self-evolving agents into representative formats. We employ an LLM-based judge to assess whether each inference attempt is successful or failed, and use different memory generation prompts for each case, following previous work (Ouyang et al., 2025; Cao et al., 2025). Detailed descriptions for each memory format are as follows. The structure illustration for each format is shown in Figure 2, and prompts used in memory generation are in Appendix J.

Trajectory In this memory representation, we concatenate all commands and codes called by the agent a_i and their execution results o_i from H without reasoning sentences r_i, and save it with the source task t. Trajectory memory M_T can be defined as $M_T = (t, [(a_1, o_1), \ldots, (a_n, o_n)])$. This contains detailed information of task solving experience even with failed steps. The agent can also infer expected outcomes by referencing observations of similar commands in memory.

Workflow In order to focus only on meaningful code snippets in the entire trajectory, this memory representation is generated by extracting a reusable workflow from the trajectory (Wang et al., 2024c). Specifically, we provide H to LLM and ask to generate a goal of workflow g and extract meaningful actions a to achieve the goal. Therefore, workflow memory M_W is denoted as $M_W = (g, [a_i, a_j, \ldots, a_k])$. By retaining only a subset of the action and observation history, Workflow is much shorter than Trajectory which reduces the risk of distractions from irrelevant information.

Summary One key principle in leveraging memory is to follow the successful actions and reflect failures from previous inference. However, raw code commands and observations do not explicitly provide the analysis why the agent succeeds or fails and the findings from the history. Thus, for Summary memory M_S, we prompt LLM to summarize the task, environment, actions, results, and analysis on why this run succeeds or fails from the trajectory. In detail, LLM generates a task summary s_t and one paragraph of experience summary s_e from the trajectory, denoted as $M_S = (s_t, s_e)$.

Insight We can reasonably expect that memory should be generalized to be easily adapted to different tasks, and as the most general memory representation, we employ the Insight format. Following the memory design of ReasoningBank (Ouyang et al., 2025), Insight M_I consists of three parts: title i_t, description i_d, content i_c, represented as $M_I = (i_t, i_d, i_c)$. In the content of this memory item, we prompt LLM to write insights on why this task is successfully or incorrectly accomplished without mentioning specific files or details. Additionally, we explicitly instruct LLM to generate generalizable insights for future similar tasks.

2.1.2 Memory Retrieval

Memory Pool Construction After finishing memory generation for all benchmarks, we construct two types of memory pool to experiment self-evolving and memory transfer learning. For the self-evolving setting, we construct a memory pool per memory type by collecting all memories generated from tasks within the testing benchmark. Formally, memory pool $\mathcal{P}$ used for self-evolving model in evaluating benchmark B_i with memory type τ is denoted as $\mathcal{P}_\tau^{\text{SE}}(B_i) = \{M_\tau^{(k)} \mid t^{(k)} \in B_i\}_{k=1}^{N_i}$. For MTL, we gather memories from all benchmarks except the testing benchmark for each

	LCB	AP	SWE	TB2	RB	ML	Avg.
Zero-shot	0.910	0.470	0.730	0.315	0.111	0.667	0.523
Self-Evolve (T)	0.930	0.470	0.730	0.326	0.133	0.750	0.536
Self-Evolve (W)	0.890	0.460	0.720	0.393	0.156	0.833	0.542
Self-Evolve (S)	0.860	0.410	0.750	0.348	0.178	0.583	0.521
Self-Evolve (I)	0.920	0.440	0.750	0.337	0.133	0.500	0.527
MTL (T)	0.940	0.490	0.770	0.270	0.122	0.583	0.534
MTL (W)	0.920	0.470	0.770	0.348	0.111	0.583	0.538
MTL (S)	0.930	0.460	0.760	0.371	0.133	0.667	0.546
MTL (I)	0.930	0.470	0.770	0.360	0.189	0.750	**0.560**
Δ	+2.0%	0.0%	+4.0%	+4.5%	+7.8%	+8.3%	+3.7%

Table 1: **Evaluation results of MTL (Pass@3).** MTL notably improves performance over both zero-shot and self-evolving settings. Among memory types, Insight achieves the highest average performance.

memory format. In formal notation, memory pool $\mathcal{P}$ used for MTL in evaluating benchmark B_i with memory type τ is $\mathcal{P}_\tau^{\text{MTL}}(B_i) = \{M_\tau^{(k)} \mid t^{(k)} \notin B_i\}_{k=1}^{N_i}$. When constructing the memory pools, we extract an embedding for each memory using a text encoder and store it alongside the memory entry.

Memory Retrieval In the inference stage, we retrieve N relevant memories for each task from the memory pool corresponding to the testing model, benchmark, and memory type, and inject retrieved memories into the system prompt of the coding agent at the beginning of the inference. In detail, we generate the embedding feature of current task and measure the cosine similarity between task embedding feature and memory embedding features. Finally, we select the final memories by top-N sampling with the highest similarity scores.

2.2 Experimental Details

Datasets We evaluate the coding agents with different memory utilization methods with 6 coding benchmarks. For competitive and function-level programming tasks, we use Aider Polyglot (Gauthier, 2024) and LiveCodeBenchv6 (LCB) (Jain et al., 2024). For repository-level coding tasks, we employ SWE-Bench Verified (SWE) (Jimenez et al., 2024) and Terminal-Bench2 (TB2) (Merrill et al., 2026). We also evaluate MTL on domain-specific coding benchmarks, such as ReplicationBench (RB) (Ye et al., 2025) for scientific knowledge grounding code generation and MLGymBench (ML) (Nathani et al., 2025) for machine learning research tasks. We randomly sample 100 tasks if the total number of samples is over 100, and we evaluate task success using evaluation protocol of each benchmark and report Pass@3 results.

Additional Details We adopt GPT-5-mini model for every LLM usage from memory generation and agent inference to LLM-based judgment. We also exploit mini-swe-agent (Yang et al., 2024) for coding agent, harbor (Team, 2026) for evaluation platform, and text-embedding-3-small model of OpenAI for text embedding extraction. In memory retrieval, we select three memories for each query ($N = 3$). More details are in Appendix C including retrieval query and benchmark descriptions.

Method	#Memories	LCB	SWE	RB	Avg.
Zeroshot	-	0.910	0.730	0.111	0.584
CER	936	0.850	0.720	0.144	0.571
ExpeL	216	0.900	0.720	0.122	0.581
ReasoningBank	97	0.920	0.750	0.133	0.601
AgentKB	5,899	0.920	0.720	0.200	0.613
MTL (I)	431	0.930	0.770	0.189	**0.630**

Table 2: **Pass@3 comparison with self-evolving methods.** MTL shows the strongest performance compared to baselines with efficient number of memories.

	LCB	AP	SWE	TB2	RB	ML	Avg.
DeepSeek V3.2							
Zero-shot	0.930	0.590	0.530	0.337	0.267	0.583	0.542
MTL (I)	0.940	0.580	0.590	0.393	0.278	0.667	**0.568**
Δ	+1.0%	-1.0%	+6.0%	+5.6%	+1.1%	+8.3%	+2.6%
Qwen3-Coder-480B-A35B-Instruct							
Zero-shot	0.800	0.460	0.590	0.292	0.211	0.583	0.483
MTL (I)	0.810	0.480	0.620	0.326	0.211	0.583	**0.501**
Δ	+1.0%	+2.0%	+3.0%	+3.4%	0.0%	0.0%	+1.8%

Table 3: **MTL results on open-source models (Pass@3).** MTL with Insight improves performance across both Deepseek V3.2 and Qwen3-Coder.

3 Experimental Results and Analysis

We now present and analyze the results of our experiments, and introduce academic findings on characteristics of Memory Transfer Learning.

3.1 Overall Performance of MTL

Main Results The results of the coding agent with Memory Transfer Learning across six coding benchmarks are shown in Table 1. MTL with Insight significantly improves the performance compared to the Zero-shot and Self-Evolve variants baselines. In particular, when Insight memories are transferred, the agent achieves more than 4.0% (up to 8.3%) performance gains on four benchmarks than Zero-shot. Insight memory transfer also outperforms the strongest Self-Evolve baseline with Workflow by 1.8% in average. These results highlight that transferable knowledge exists across different domains, and leveraging such knowledge is crucial for enhancing coding agents.

Comparison with Self-Evolving Approaches We further evaluate MTL against four existing self-evolving baselines adapted to construct and retrieve coding-domain memory under the same setup, reporting Pass@3. As presented in Table 2, MTL

Model	Zero-shot	Memory Transfer Learning with Insight
Memory	(No Memory)	Memory Item 2 (Generated from LiveCodeBenchv6): **Title**: Create quick self-contained tests using an inline Python here-doc to validate fixes **Description**: When making small code fixes, write a minimal tests... **Content**: Set up a short battery of tests that cover ...
Reasoning	I will modify django/db/models/aggregates.py to raise a FieldError when an Aggregate's source expression contains a window (over) clause. This prevents creating an aggregate over a window expression ...	I will use Memory Item 2 (use an inline Python here-doc for safe, atomic edits and quick verification) to modify django/db/models/sql/query.py. The goal is to ensure that when resolving aggregates we detect if any referenced annotation either is a subquery or contains a window expression ...
Result	Fail	Success

Table 4: **Case Study: Zero-shot vs. Memory Transfer Learning with Insight.** Transferred Insight from LiveCodeBenchv6 to SWE-Bench Verified provides meta-knowledge regarding strategic guidance (*e.g.,* inline test validation), allowing the agent to succeed where the zero-shot baseline fails.

outperforms the two strongest baselines, ReasoningBank (Ouyang et al., 2025)and AgentKB (Tang et al., 2025), by +2.9% and +1.7%, respectively, demonstrating that its performance gains are substantial even relative to strong self-evolving baselines. In particular, ReasoningBank achieves a limited average gain, as it only leverages a small number of in-domain memories and does not utilize cross-domain knowledge. In contrast, AgentKB leverages a large number of out-of-domain memories (from general reasoning tasks), but still underperforms our method despite using around 5.8k memories. Notably, MTL uses only 431 memories in the memory pool, yet achieves the highest average performance. This highlights both the effectiveness and efficiency of our approach compared to existing self-evolving methods. On the other hand, CER (Liu et al., 2025b) and ExpeL (Zhao et al., 2024), originally proposed for non-coding tasks, fail to surpass the zero-shot baseline when adapted to coding-domain memory, suggesting that general-purpose memory frameworks do not directly transfer to coding settings.

Model-Agnostic Applicability These results are further validated across different models. We evaluate the effectiveness of MTL on the DeepSeek V3.2 (Liu et al., 2025a) and Qwen3-Coder-480B-A35B-Instruct (Yang et al., 2025; Cao et al., 2026) models, achieving average performance improvements of 2.6% and 1.8%, respectively. Notably, these results indicate that our method is also beneficial for open-source models, highlighting the broad applicability of cross-domain memory transfer.

Core Finding 1. *MTL significantly improves coding agent performance and outperforms self-evolving methods in effectiveness and efficiency.*

3.2 Mechanism of Memory Transfer Learning

How Does Memory Transfer Learning Benefit the Agents? To investigate the operational mechanisms of MTL, we inspect the inference outcomes

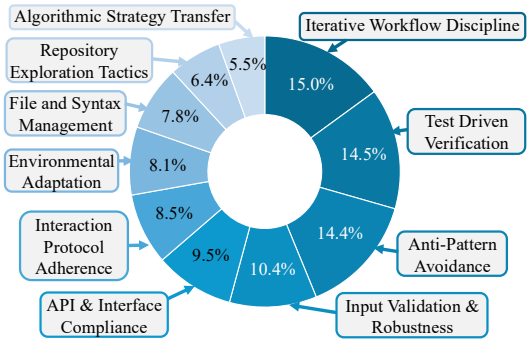

Figure 3: **Memory Transfer Contribution Breakdown.** Meta-knowledge drives most transfer gains.

using LLM and manual case studies. First, we collect trajectories where the agent fails in zero-shot but succeeds when MTL with Insight memory is applied, and use GPT-5 to categorize how transferred memory contributes to successful task completion (see Appendix I for predefined categories). As presented in Figure 3, our analysis reveals that transferred memory primarily benefits agents by providing meta-knowledge rather than task-specific content. This meta-knowledge includes structured action workflow (e.g., inspect, edit, verify, submit), guardrails for compliance with external constraints (such as output formats, function signatures, and API contracts), and disciplined programming practices that discourage large one-shot refactors, blind overwrites, and brittle hardcoding. Transferred memory further promotes risk-controlled editing through minimal patch strategies, self-generated verification when official tests are unavailable, and safe interaction with execution environments by anticipating tool-chain and infrastructure failures. On the other hand, *Algorithmic Strategy Transfer* accounts for only 5.5% of the total gains in Figure 3, suggesting that direct algorithmic transfer of specific codes or algorithms remains limited.

Case Study: Zero-shot vs. MTL The effect of MTL and transferred meta-memory are also shown in the case study. In Table 4, we compare the inference outcomes between zero-shot and MTL tested

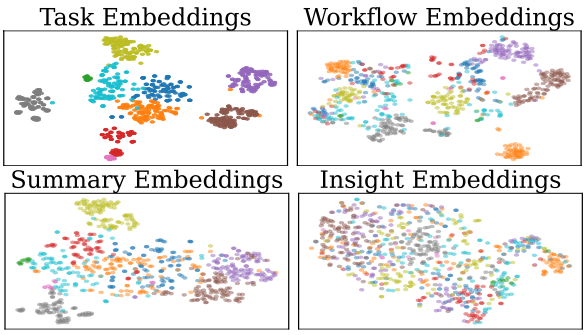

Figure 4: **t-SNE Visualization of Memory Formats.** Colors indicate benchmarks; Insight features are sparse and intermingled, indicating their task-agnostic nature.

on one instance of SWEBench-Verified. In the zero-shot setting, the model naively solves the task by simply raising an error and eventually fail the test. However, with MTL, retrieved Insight generated from LiveCodeBenchv6 provides behavior knowledge about testing with an inline Python here-doc to validate fixes, and the agent follows the guideline of it and successfully completes the task. This highlights the practical impact of transferred meta-memory in enabling successful completion.

Core Finding 2. *Transferable knowledge exists across distinct task types, and its primary form is meta-memory encoding procedural and behavioral guidance, not domain-specific knowledge.*

3.3 Impact of Memory Abstraction

Abstraction Level of Four Memory Types We adopt four memory representations in our experiments, each designed with a distinct level of abstraction. Trajectory and Workflow memories are less abstract and highly task-specific, containing raw command-level actions, whereas Summary and Insight memories are more abstract and generalized. These properties are clearly reflected in the embedding space visualizations shown in Figure 4. Task embeddings form benchmark-level clusters, while embeddings in the Insight space become increasingly sparse and intermingled across benchmarks, indicating that Trajectory and Workflow remain task-specific, whereas Summary and Insight exhibit greater generality. In addition, memory embedding distributions are quantitatively characterized using the Davies–Bouldin Index (DBI) and the Local Inverse Simpson's Index (LISI), as shown in Figure 5. The increasing DBI values indicate progressively weaker benchmark-level cluster separation, while the increasing LISI values indicate stronger local benchmark mixing, supporting the transition from task-specific to generalized memory.

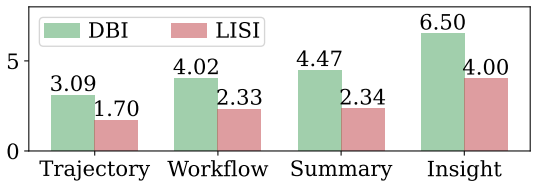

Figure 5: **Embedding Space Analysis,** showing weaker separation and stronger mixing at higher abstraction.

Method	LCB	SWE	RB	Avg.
Task-specific Insights	0.887	0.617	0.067	0.523
Task-agnostic Insights	0.893	0.627	0.082	**0.534**
Δ	+0.6%	+1.0%	+1.5%	+1.1%

Table 5: **Abstraction Effect.** The superiority of task-agnostic insights highlights abstraction as a key driver.

Correlation between Abstraction and Transfer Effectiveness In Table 1, under the MTL setting, the Insight format achieves the best performance, followed by Summary, Workflow, and Trajectory, while all MTL variants outperform the zero-shot baseline. As discussed in Section 3.2, transferred memories primarily provide general meta-knowledge, whereas implementation details from unrelated tasks may distract the agent. Thus, more abstract and generalized memory representations tend to yield higher transfer effectiveness. We further validate the relationship between memory abstraction level and transferability through a case study comparing memory transfer between Trajectory and Insight memories in Appendix F.

Isolating the Effect of Memory Abstraction To isolate the effect of abstraction, we compare two groups of memories within the same representation (Insight): relatively task-specific and task-agnostic memories. Specifically, we prompt an LLM to infer the original task solely from each Insight memory and measure the similarity between the inferred task and ground-truth task. Higher similarity indicates that the memory retains more task-specific information, while lower similarity indicates that it is more abstract and task-agnostic. Based on this measure, we partition the memories into the top 30% (task-specific) and bottom 30% (task-agnostic). In Table 5, we observe that within the same memory format, task-agnostic memories consistently outperform task-specific ones. This indicates that abstraction, rather than format itself, is a key factor driving transfer performance. We also present the formal modeling on the correlation of memory abstraction and transfer effectiveness in Appendix H.

Core Finding 3. *More abstract and general memory representations yield higher transfer effectiveness by avoiding brittle implementation anchoring.*

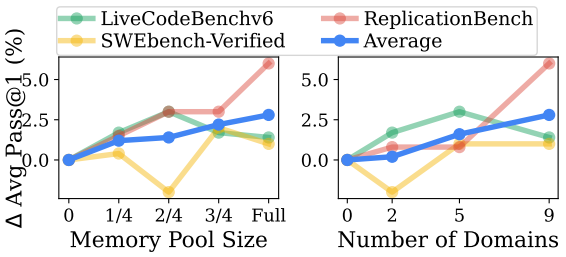

Figure 6: **Memory Scaling.** Larger memory pools and more domains yield better performance.

Source	Target	LCB	SWE	RB	Avg.
Zeroshot	GPT-5-mini	0.863	0.623	0.059	0.515
DeepSeek V3.2		0.890	0.617	0.048	0.518
Qwen3-Coder		0.883	0.607	0.093	0.528
GPT-5-mini		0.877	0.633	0.119	**0.543**
Zeroshot	DeepSeek V3.2	0.890	0.423	0.144	0.486
GPT-5-mini		0.890	0.450	0.163	0.501
DeepSeek V3.2		0.893	0.463	0.178	**0.511**
Zeroshot	Qwen3-Coder	0.733	0.347	0.126	0.402
GPT-5-mini		0.780	0.347	0.111	**0.413**
Qwen3-Coder		0.740	0.370	0.130	**0.413**

Table 6: **Cross-Model Memory Transfer.** Average Pass@1 results show consistent gains over zero-shot.

3.4 Further Analysis and Ablations

Negative Transfer in MTL As shown in Table 1, MTL can degrade performance in certain cases. To understand this, we analyzed instances where zero-shot setting succeeded but MTL failed. We categorized these negative transfer cases as follows:

- **Domain-mismatched anchoring**: Structurally irrelevant but superficially similar memories act as misleading anchors. These introduce incorrect assumptions, diverting the agent's reasoning from core logic and constraints.
- **False validation confidence**: Verification memories can create a false sense of certainty. This leads to self-confirming loops where agents rely on superficial checks, resulting in missed specifications and silent failures.
- **Misapplied best-practice transfer**: Successful patterns are sometimes transferred indiscriminately, overriding task-specific semantics and leading to rigid adherence to familiar workflows that violate new task requirements.

These failure modes stem from wrong memory retrieval and failed adaptation to the target task. Thus, performance degradation can be mitigated by retrieving truly helpful, not just semantically relevant, memories and through better adaptation methods such as memory rewriting (Cao et al., 2025). Case studies in Appendix G further shows that negative transfer arises when retrieved memories are blindly imitated or misread, causing agents to apply incompatible patterns or justify weak shortcuts.

Finding 4. *Negative memory transfer mainly arises from misleading anchors, false validation signals, and misapplied procedural reuse.*

Impact of the Memory Pool Size To investigate how MTL scales with memory availability and diversity, we evaluate our method by varying both the memory pool size and the number of source domains across three benchmarks. For memory pool size, we randomly sample memories from the full cross-domain memory pool at ratios of 1/4, 2/4, and 3/4 of the original size. As shown in Figure 6, the average performance consistently improves as the number of memories increases, suggesting that larger memory pools increase the likelihood of retrieving relevant memories for the target task. We further examine scalability with respect to the number of source domains and find that the average performance gain generally increases as more domains are incorporated, with 9 domains yielding the best performance. These results demonstrate that MTL also benefits from broader domain coverage, as greater memory diversity provides more transferable meta-knowledge and improves the chance of retrieving useful knowledge.

Finding 5. *The effectiveness of MTL scales with memory pool size and domain coverage.*

Cross-model Memory Transfer Learning To validate whether memories are transferable across models, we evaluate agent performance under MTL using memories generated by different models. We hypothesize that if MTL mainly benefits from meta-knowledge, then memories from different models should also be effective, as such meta-knowledge is not model-specific but instead relates to the testing environment and general coding guidelines. As shown in Table 6, all cross-model MTL variants consistently outperform the zero-shot baseline. In particular, cross-model memory transfer is effective in both directions, from stronger to weaker models and vice versa. These findings support our hypothesis that meta-knowledge is transferable across models. Yet, cross-model transfer generally underperforms MTL with self-generated memories, suggesting that model-specific biases may exist.

Finding 6. *Memory can be transferred across models, while native memories yield the best results.*

4 Related Work

4.1 Coding Agents

As LLM have demonstrated strong capabilities in code generation (Roziere et al., 2023; Hui et al.,

2024; Zhu et al., 2024), researchers have developed LLM-based coding agents that interact with programming environments such as bash shells (Team, 2026; Wang et al., 2024a) through diverse systematic designs, and have evaluated them across a wide range of coding tasks. At the early stage of coding agents, they target function-level code generation tasks (Chou et al., 2025; Jain et al., 2024; Xia et al., 2024) in a single file. AlphaCodium (Ridnik et al., 2024) proposed a flow engineering in code generation which iteratively run reasoning, generation, ranking, and debugging. LDB (Zhong et al., 2024) introduced a novel debugging framework with language models that leverage runtime execution information for function-level code generation. Beyond a single file level editing, CodeAgent (Zhang et al., 2024), RepoAgent (Luo et al., 2024), RL-Coder (Wang et al., 2024b) address repository-level code modification tasks (Jimenez et al., 2024; Merrill et al., 2026). Furthermore, code agents targeting domain-specific tasks, such as Paper2Code (Seo et al., 2025) for code generation for ML paper replication tasks and BixbBench (Mitchener et al., 2025) for computational biology related tasks.

4.2 Memory-based Self-Evolving Agents

Self-evolving agents (Cai et al., 2025) leverage past experiences by reusing successful solution patterns in similar tasks and avoiding previously encountered erroneous actions. To manage these experiences effectively, existing memory-based self-evolving agents (Yeo et al., 2025b; Kim et al., 2026) primarily focus on mechanisms for memory generation and retrieval during interactions with environments. (Fang et al., 2025b; Chen et al., 2025) AWM (Wang et al., 2024c) proposed memory utilization through the collection of common workflows in web agents, while ReasoningBank (Ouyang et al., 2025) extracts helpful insights from trajectories via test-time scaling. Dynamic Cheatsheet (Suzgun et al., 2025) constructs evolving memories that encode reusable strategies and insights, and ReMe (Cao et al., 2025) presents a holistic framework from memory generation to retrieval and memory refinement. MemEvolve (Zhang et al., 2025) further introduces system-level evolution through meta-evolution in memory agents. However, existing memory-based self-evolving agents are primarily evaluated within the same benchmark or task domain, overlooking the potential value of memories generated from other task domains.

4.3 Transfer Learning

Transfer Learning (Zhuang et al., 2020) has been extensively studied as the reuse of knowledge acquired in a source domain to improve performance in a target domain. Traditional approaches mainly rely on parametric adaptation through model updates (Howard and Ruder, 2018; Houlsby et al., 2019). With the emergence of LLMs demonstrating strong generalization capabilities, recent work has increasingly explored non-parametric knowledge transfer mechanisms. In-context learning (Dong et al., 2024; Min et al., 2022; Kim et al., 2025), as a representative paradigm, shows that LLMs can reuse knowledge provided in the context at inference time. In the agent setting, knowledge is instead generated by the model itself in the form of memory and transferred across tasks. AgentKB (Tang et al., 2025) introduces a framework for managing and leveraging a unified memory pool across multiple task domains. However, prior work lacks deeper analysis of transfer mechanisms and constructs unified memory spaces across heterogeneous environments, missing the opportunity to exploit coding-specific shared principles.

5 Conclusion

In this work, we presented the first holistic investigation into Memory Transfer Learning for coding agents, challenging the prevailing assumption that memory utilization must be limited to homogeneous task domains. Through extensive evaluation across 6 diverse benchmarks, we demonstrated that leveraging a unified memory pool from heterogeneous domains can enhance agent performance by 3.7%. Our analysis yields three critical design principles for cross-domain memory. First, we identified that the primary value of transferred memory lies in meta-knowledge rather than task-specific workflows. Second, we found that abstraction dictates transferability; high-level abstractions like Insights generalize effectively across domains, whereas low-level Trajectories often induce negative transfer due to brittle implementation anchoring. Third, we highlighted that the effectiveness of memory transfer scales with the size and diversity of the memory pool, increasing the likelihood of retrieving useful meta-knowledge. We hope this study establishes empirical foundations for expanding memory utilization beyond a single domain and stimulates further research into robust memory usage strategies for self-evolving coding agents.

Limitations

This work discuss about the effectiveness of memory transfer across heterogeneous domains in coding agents, however, there could be several limitations in this study. First, since our research scope is focused on coding agents, the applicability of our findings into other agent settings, such as web agents or reasoning agents, should be evaluated separately before use. Moreover, the specific memory formats (Trajectory, Workflow, Summary, Insight) examined in this work may not generalize to other agent types that operate with fundamentally different action spaces or knowledge structures. Second, while we analyze the scalability of MTL in Figure 6, MTL is not evaluated in a real-world setting with a massive number of memories accumulated from numerous users, leaving the practical scalability of MTL in production environments an open question. Third, although we provide extensive evaluation across six benchmarks, the benchmarks used in this work are all English-based, and the transferability of MTL to non-English or multilingual coding environments remains unexplored. Lastly, although MTL lowers the barriers to developing versatile software engineering agents, however, we acknowledge the potential for negative transfer, where agents might misapply implementation patterns or overlook domain-specific safety constraints. Consequently, the deployment of such systems requires careful attention to robust retrieval strategies to prevent the generation of unreliable or insecure code.

Ethical Considerations

We identify several ethical considerations that users should be aware of in real-world deployments. Since memories are automatically generated by LLMs without human verification, they may contain incorrect, biased, or hallucinated content that can propagate across tasks through memory transfer, potentially amplifying errors at scale. Furthermore, as the memory pool grows with contributions from numerous users in real-world deployments, it may inadvertently capture and retain sensitive information embedded in user task descriptions or execution traces, raising privacy concerns. To mitigate these risks, we recommend employing additional memory verification and safeguard mechanisms to ensure the reliability and privacy of the memory pool prior to deployment.

References

Zhicheng Cai, Xinyuan Guo, Yu Pei, Jiangtao Feng, Jinsong Su, Jiangjie Chen, Ya-Qin Zhang, Wei-Ying Ma, Mingxuan Wang, and Hao Zhou. 2025. Flex: Continuous agent evolution via forward learning from experience. *arXiv preprint arXiv:2511.06449*.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, and 1 others. 2026. Qwen3-coder-next technical report. *arXiv preprint arXiv:2603.00729*.

Zouying Cao, Jiaji Deng, Li Yu, Weikang Zhou, Zhaoyang Liu, Bolin Ding, and Hai Zhao. 2025. Remember me, refine me: A dynamic procedural memory framework for experience-driven agent evolution. *arXiv preprint arXiv:2512.10696*.

Silin Chen, Shaoxin Lin, Xiaodong Gu, Yuling Shi, Heng Lian, Longfei Yun, Dong Chen, Weiguo Sun, Lin Cao, and Qianxiang Wang. 2025. Swe-exp: Experience-driven software issue resolution. *arXiv preprint arXiv:2507.23361*.

Jason Chou, Ao Liu, Yuchi Deng, Zhiying Zeng, Tao Zhang, Haotian Zhu, Jianwei Cai, Yue Mao, Chenchen Zhang, Lingyun Tan, Ziyan Xu, Bohui Zhai, Hengyi Liu, Speed Zhu, Wiggin Zhou, and Fengzong Lian. 2025. Autocodebench: Large language models are automatic code benchmark generators. *Preprint*, arXiv:2508.09101.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. 2024. A survey on in-context learning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 1107–1128, Miami, Florida, USA. Association for Computational Linguistics.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, and 1 others. 2025a. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. 2025b. Memp: Exploring agent procedural memory. *arXiv preprint arXiv:2508.06433*.

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

Paul Gauthier. 2024. Aider polyglot benchmark. Blog post and benchmark details.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for nlp. In *International conference on machine learning*, pages 2790–2799. PMLR.

Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Kangsan Kim, Geon Park, Youngwan Lee, Woongyeong Yeo, and Sung Ju Hwang. 2025. Videoicl: Confidence-based iterative in-context learning for out-of-distribution video understanding. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 3295–3305.

Kangsan Kim, Yanlai Yang, Suji Kim, Woongyeong Yeo, Youngwan Lee, Mengye Ren, and Sung Ju Hwang. 2026. Ma-egoqa: Question answering over egocentric videos from multiple embodied agents. *arXiv preprint arXiv:2603.09827*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025a. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Yitao Liu, Chenglei Si, Karthik R Narasimhan, and Shunyu Yao. 2025b. Contextual experience replay for self-improvement of language agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14179–14198.

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, and 1 others. 2024. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, and 1 others. 2026. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*.

Ludovico Mitchener, Jon M Laurent, Alex Andonian, Benjamin Tenmann, Siddharth Narayanan, Geemi P Wellawatte, Andrew White, Lorenzo Sani, and Samuel G Rodriques. 2025. Bixbench: a comprehensive benchmark for llm-based agents in computational biology. *arXiv preprint arXiv:2503.00096*.

Deepak Nathani, Lovish Madaan, Nicholas Roberts, Nikolay Bashlykov, Ajay Menon, Vincent Moens, Amar Budhiraja, Despoina Magka, Vladislav Vorotilov, Gaurav Chaurasia, and 1 others. 2025. Mlgym: A new framework and benchmark for advancing ai research agents. *arXiv preprint arXiv:2502.14499*.

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

Chen Qian, Yufan Dang, Jiahao Li, Wei Liu, Zihao Xie, Yifei Wang, Weize Chen, Cheng Yang, Xin Cong, Xiaoyin Che, and 1 others. 2024. Experiential co-learning of software-developing agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5628–5640.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. 2025. Paper2code: Automating code generation from scientific papers in machine learning. *arXiv preprint arXiv:2504.17192*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. 2025. Dynamic cheatsheet: Test-time learning with adaptive memory. *arXiv preprint arXiv:2504.07952*.

Xiangru Tang, Tianrui Qin, Tianhao Peng, Ziyang Zhou, Daniel Shao, Tingting Du, Xinming Wei, Peng Xia, Fang Wu, He Zhu, Ge Zhang, Jiaheng Liu, Xingyao Wang, Sirui Hong, Chenglin Wu, Hao Cheng, Chi Wang, and Wangchunshu Zhou. 2025. Agent KB: leveraging cross-domain experience for agentic problem solving. *arXiv preprint arXiv:2507.06229*.

Harbor Framework Team. 2026. Harbor framework: A framework for evaluating and optimizing agents and models in container environments. https://github.com/laude-institute/harbor.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024a. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024b. Rlcoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487*.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2024c. Agent workflow memory. *arXiv preprint arXiv:2409.07429*.

Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. 2024. Top leaderboard ranking = top coding proficiency, always? evoeval: Evolving coding benchmarks via llm. *arXiv preprint*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Christine Ye, Sihan Yuan, Suchetha Cooray, Steven Dillmann, Ian LV Roque, Dalya Baron, Philipp Frank, Sergio Martin-Alvarez, Nolan Koblischke, Frank J Qu, and 1 others. 2025. Replicationbench: Can ai agents replicate astrophysics research papers? *arXiv preprint arXiv:2510.24591*.

Woongyeong Yeo, Kangsan Kim, Soyeong Jeong, Jinheon Baek, and Sung Ju Hwang. 2025a. Universalrag: Retrieval-augmented generation over corpora of diverse modalities and granularities. *arXiv preprint arXiv:2504.20734*.

Woongyeong Yeo, Kangsan Kim, Jaehong Yoon, and Sung Ju Hwang. 2025b. Worldmm: Dynamic multimodal memory agent for long video reasoning. *arXiv preprint arXiv:2512.02425*.

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

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19632–19642.

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and 1 others. 2025. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. 2024. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2020. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76.

A Average Pass@1 Performance of MTL

Table 8 reports Pass@1 scores averaged over three runs as a complementary metric to the main Pass@3 results. The findings remain consistent: MTL consistently outperforms the zero-shot baseline, with the Insight format achieving the best overall performance. The low standard deviation across runs further confirms the reliability and reproducibility of our evaluation.

B Analysis on Retrieval Methods

As discussed in Section 3.4, negative transfer often arises from incorrect memory retrieval and adaptation. We therefore investigate whether advanced retrieval strategies, such as reranking and memory rewriting, can further improve MTL. For reranking, we first retrieve 20 candidate memories based on embedding similarity and then prompt the LLM to select the three most helpful ones for the given task. For task-adaptive memory rewriting, we prompt the LLM to rewrite the retrieved memories to better align with the target task. However, both methods underperform simple embedding-based retrieval, as shown in Table 7. This is likely because the required knowledge is difficult to anticipate in dynamic, multi-step agent settings. These findings suggest that retrieval methods designed for static settings may not generalize well to cross-domain memory transfer, highlighting the need for further study on agentic memory retrieval and adaptation, such as domain routing (Yeo et al., 2025a) and step-wise memory retrieval (Cao et al., 2025).

Method	LCB	SWE	RB	Avg
No Memory	0.910	0.730	0.111	0.584
LLM Reranking	0.920	0.730	0.144	0.598
Adaptive Rewriting	0.920	0.760	0.144	0.608
Embedding Similarity	0.930	0.770	0.189	**0.630**

Table 7: **Retrieval Method Comparison** Pass@3 results show that simple embedding-based retrieval outperforms advanced methods.

C Additional Implementation Details

C.1 Retrieval queries

In querying Trajectory memory, we use embedding similarities between target task and tasks in the memories, since both query and memories have the *Task* information. In querying other memories (Workflow, Summary, and Insight), which do not have *Task* information, we ask the model to write 4-5 sentences of coding plan to solve the given task and use the plan as the query.

C.2 Benchmarks

We provide brief descriptions of the six coding benchmarks used in our evaluation.

LiveCodeBenchv6 (Jain et al., 2024) is a contamination-free benchmark for function-level code generation, continuously collecting new problems from competitive programming platforms such as LeetCode, Codeforces, and AtCoder. We use version 6, sampling 100 problems and evaluating in Python.

Aider-Polyglot (Gauthier, 2024) evaluates code editing across multiple programming languages, including Python, JavaScript, TypeScript, Rust, Go, and C++. Tasks require agents to modify existing files to implement requested changes, testing their ability to operate across polyglot codebases.

SWE-Bench Verified (Jimenez et al., 2024) is a repository-level benchmark where agents must resolve real-world GitHub issues from popular Python repositories. The Verified split contains 500 human-validated instances, requiring agents to localize and fix bugs across large codebases. We sample 100 problems in our evaluation.

Terminal-Bench2 (Merrill et al., 2026) evaluates agents on 89 hard, realistic tasks in command-line interfaces, including system-level operations, file manipulation, and shell scripting, testing their ability to interact robustly with Unix-style execution environments.

ReplicationBench (Ye et al., 2025) consists of 90 tasks that test whether AI agents can replicate scientific results from astrophysics research papers by generating the corresponding code and reproducing the reported findings, requiring grounding in domain-specific scientific knowledge.

MLGym-Bench (Nathani et al., 2025) comprises 12 machine learning research tasks, including model improvement, hyperparameter tuning, and ML pipeline development, where agents are expected to explore and implement strategies that outperform a given baseline.

C.3 Usage of AI Assistants

We utilize AI assistants to support experimental code development, related work survey, and paper writing. All outputs generated by AI assistants were carefully reviewed and verified by the authors to ensure the accuracy and integrity of the reported results and written content.

	LiveCodeBenchv6	Aider-Polyglot	SWEBench-Verified	TerminalBench2	ReplicationBench	MLGym-Bench	Avg.
GPT-5-mini							
ZeroShot	0.863±0.005	0.343±0.009	0.623±0.009	0.206±0.019	0.059±0.014	0.583±0.068	0.435
MTL (T)	0.890±0.008	0.357±0.026	0.610±0.016	0.195±0.009	0.063±0.005	0.528±0.039	0.438
MTL (W)	0.877±0.026	0.350±0.014	0.620±0.014	0.243±0.035	0.081±0.014	0.583±0.000	0.449
MTL (S)	0.887±0.012	0.370±0.008	0.613±0.009	0.228±0.028	0.078±0.009	0.611±0.040	0.451
MTL (I)	0.877±0.012	0.347±0.009	0.633±0.034	0.213±0.027	0.119±0.032	0.639±0.104	0.454
Δ	+1.3%	+0.3%	+1.0%	+0.8%	+5.9%	+5.6%	+1.9%
Qwen3-Coder-480B-A35B-Instruct							
ZeroShot	0.733±0.019	0.357±0.021	0.347±0.021	0.210±0.005	0.126±0.028	0.500±0.000	0.366
MTL (I)	0.740±0.024	0.360±0.014	0.370±0.022	0.228±0.035	0.130±0.014	0.528±0.039	0.377
Δ	+0.7%	+0.3%	+2.3%	+1.9%	+0.4%	+2.8%	+1.2%
DeepSeek V3.2							
ZeroShot	0.890±0.008	0.433±0.026	0.423±0.017	0.266±0.043	0.144±0.009	0.500±0.068	0.446
MTL (I)	0.893±0.005	0.447±0.012	0.463±0.012	0.288±0.014	0.178±0.000	0.556±0.079	0.466
Δ	+0.3%	+1.3%	+4.0%	+0.4%	+3.3%	+5.6%	+2.0%

Table 8: **Average of Pass@1 with standard deviation results of Memory Transfer Learning acros three runs.**

C.4 Usage of Existing Artifacts

All benchmarks we used are open-sourced and we strictly follow the original intended use for each benchmark. All datasets used in this work are publicly available benchmarks that do not contain any personally identifying information or offensive content. All artifacts used in this work are publicly available and permit use for research purposes.

	# Avg. Tokens		
Benchmark	**Total Input**	**Memory**	**Ratio**
LiveCodeBenchv6	6,772	578	8.54%
Aider-Polyglot	18,633	586	3.15%
SWEBench-Verified	178,407	528	0.30%
Terminal-Bench2	258,571	618	0.24%
ReplicationBench	107,544	615	0.57%
MLGym-Bench	67,068	613	0.91%
Average	**109,847**	**+584**	**0.53%**

Table 9: **Token Overload.** Memory tokens account for only 0.53% of total input tokens on average.

Model	**Avg Agent Exec (s)**	**Avg Retrieval (s)**	**Ratio**
GPT-5-mini	175.90	8.97	5.10%
DeepSeek V3.2	319.15	9.29	2.91%
Qwen3-Coder-480B	131.37	3.83	2.92%
Average	**208.80**	**7.36**	**3.53%**

Table 10: **Latency Overload.**

D Token and Latency Overhead of MTL

Token Overhead Analysis To validate the efficiency of Memory Transfer Learning, we present the token overload introduced by memory injection across six benchmarks in Table 9. Since the memory scheme is fixed, the number of tokens of retrieved memories remains consistently around 600 tokens across all benchmarks. However, the total input token length varies substantially across benchmarks, ranging from 6,722 tokens in LiveCodeBenchv6 to 258,571 tokens in Terminal-Bench2, reflecting the complexity and interaction lengths. As a result, the proportion of memory tokens relative to total input tokens decreases as task complexity increases, with memory overhead ranging from 8.54% on LiveCodeBenchv6 to as low as 0.24% on Terminal-Bench2. On average, memory injection accounts for only 0.53% of total input tokens, demonstrating that MTL introduces negligible token overhead regardless of benchmark complexity.

Latency Overhead Analysis We additionally measure the latency overhead introduced by the memory retrieval step in Table 10. For each of three backbone models, we sample tasks across the six evaluation benchmarks and time the end-to-end memory retrieval used by MTL. We then compare it against the agent execution time aggregated from the corresponding zero-shot runs of the same model.

Across all three backbones, memory retrieval accounts for only a small fraction of the total task-solving time. Retrieval takes between 3.83s and 9.29s on average, while agent execution itself is one to two orders of magnitude larger, ranging from 131.37s to 319.15s. The resulting retrieval-to-execution ratio stays between 2.91% and 5.10%, with an overall average of only 3.53%. This mirrors the token overhead analysis: regardless of the backbone, the cost of leveraging cross-domain memory is dwarfed by the agent's own reasoning and tool-use time, confirming that MTL is practical to deploy without a meaningful latency penalty.

E Benchmark-wise Transfer Analysis

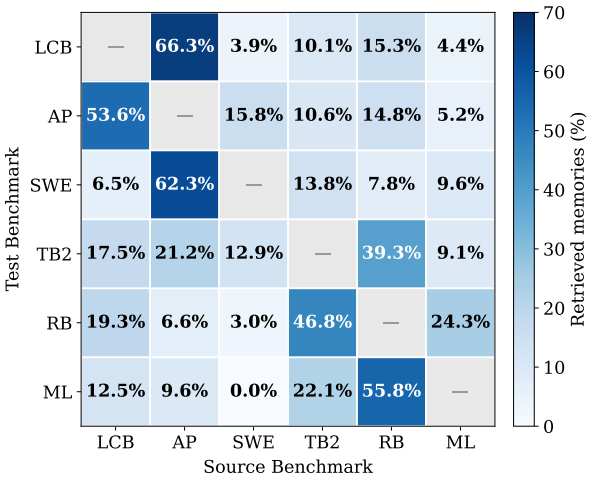

Figure 7: **Transfer Matrix of MTL.** Proportion of retrieved memories from each source benchmark for every testing benchmark.

We presents the distribution of retrieved memory sources for each testing benchmark under MTL with Insight memory in Figure 7. Two prominent transfer clusters emerge: function-level benchmarks (LiveCodeBenchv6 and Aider-Polyglot) exhibit strong mutual retrieval, with LiveCodeBenchv6 drawing 66.3% of its memories from Aider-Polyglot and vice versa (53.6%), reflecting their shared task structure in single-file code generation. Similarly, benchmarks involving complex, multi-step execution (ReplicationBench, Terminal-Bench2, and MLGym-Bench) form a tightly coupled cluster, with ReplicationBench and Terminal-Bench2 exchanging nearly half of their retrieved memories with each other (46.8% and 39.3%, respectively). In contrast, SWE-Bench Verified contributes minimally as a memory source across most benchmarks, suggesting that repository-level debugging experiences are less semantically aligned with other coding domains. These patterns indicate that even Insight memories, despite being the most abstract representation, retain residual task-specific signals that drive retrieval toward structurally similar benchmarks. This suggests that developing truly domain-agnostic memory representations that enable more uniform cross-benchmark retrieval remains a promising direction for further improving MTL effectiveness.

F Case Study on Memory Abstraction and Transfer Effectiveness

Table 11 compares representative examples of memory transfer using Trajectory and Insight format. In the Trajectory example, the agent blindly follows the commands-level instructions in the memory. This behavior is inherently risky, since the details differ across tasks and environments, including code languages and file structures. As a result, the transferred Trajectory acts as a brittle anchor, causing the agent to execute invalid commands and runtime errors. In contrast, the transferred Insight memory provides high-level behavioral guidance, such as prioritizing inspection of evaluation criteria and improving data utilization. Rather than imposing concrete implementation details that may conflict with the new task, it supplies abstract procedural principles that guide the agent's reasoning without constraining its adaptation process. As shown in its reasoning, the agent internalizes these coding practices while deriving task-specific implementation details, leading to successful task completion.

G Case Study on Negative Transfer

While Memory Transfer Learning generally improves performance, it also introduces the risk of negative transfer through blind imitation or misinterpretation of transferred knowledge. As illustrated in Table 12, we identify two primary failure modes that hinder effective transfer. First, the misapplication of technical patterns occurs when an agent incorrectly projects language-specific logic (*e.g.,* R-language file-writing routines) onto an incompatible environment like C++, leading to structural failures. Second, semantic distortion occurs when a strategic insight intended for rigorous validation is misread as a justification for suboptimal shortcuts.

H Formal Modeling of Abstraction

To formally ground our empirical findings, we introduce a mathematical framework modeling the abstraction-transfer tradeoff. We decompose a memory embedding $e(m)$ into a domain-invariant component (meta-knowledge, z_{inv}) and a domain-specific component (z_{sp}):

$$e(m) = z_{\text{inv}}(m) + z_{\text{sp}}(m).$$

We define the Abstraction level (A) of a memory as the proportion of the domain-invariant component:

$$A = \frac{\|z_{\text{inv}}(m)\|^2}{\|z_{\text{inv}}(m)\|^2 + \|z_{\text{sp}}(m)\|^2}.$$

Higher A indicates that the memory is dominated by transferable meta-knowledge rather than domain-specific details.

For an unseen target task x, the utility $U(x, m)$ of retrieving memory m is modeled as a trade-off between transferable guidance and brittle domain mismatch:

$$U(x, m) \propto \underbrace{\langle e(x), z_{\text{inv}}(m) \rangle}_{\text{Transferable Guidance}} - \underbrace{\langle e(x), z_{\text{sp}}(m) \rangle}_{\text{Domain Mismatch Penalty}} .$$

To analyze cross-domain transfer, we formalize two natural assumptions: (1) embeddings have a bounded capacity (e.g., normalized norm), meaning an increase in A strictly replaces domain-specific details with meta-knowledge, and (2) for an unseen task x, the domain-specific component z_{sp} acts as misaligned noise. Therefore, as A increases, the expected mismatch penalty decreases, allowing the universally applicable meta-knowledge (z_{inv}) to dominate the utility.

Proposition 1 (Abstraction–Transfer Trade-off). Under these assumptions, our formal model proves that the expected empirical transfer gain strictly increases with the abstraction level A.

I Memory Benefit Category

In Table 13, we present the categories of memory contributions generated by the LLM for analysis.

J Memory Generation Prompts

We present the prompts for memory generation for each memory type from Figure 8 to Figure 13.

Model	Memory Transfer Learning with Trajectory	Memory Transfer Learning with Insight
Memory	Memory Item 1 (Generated from MLGym-Bench): **Task**: You are given the train, validation and test sets of Housing Price Prediction dataset and a baseline... **Trajectory**: **Agent**: `ls -la data && nl -ba evaluate.py | sed -n '1,240p'` **System**: `<returncode>0</returncode><output>total 464 drwxr-xr-x 2 root root 4096 Dec 25 18:50 .drwxrwxrw- 1 agent agent 4096 Jan 14 06:02 ...` **Agent**: `cat «'PY' > train_predict.py ... ('onehot', OneHotEncoder(..., sparse=False)) ... PY` **System**: `TypeError: OneHotEncoder.__init__() got an unexpected keyword argument 'sparse'` ...	Memory Item 1 (Generated from MLGym-Bench): **Title**: Inspect eval requirements, combine train+val, use robust preprocessing, and adapt to library API changes **Description**: Before improving a baseline model, read the repository/evaluation expectations, train on all available labeled data, apply robust preprocessing for numeric and categorical ... **Content**: First inspect the code that loads labels and evaluates predictions so you know the exact submission format and metrics to optimize. For the final model, combine training ...
Reasoning	I'll start by inspecting the repository root... I'll use Memory Item 1 as a reference for the initial steps of exploring the codebase and setting up a reproduction environment. (Blindly following the trajectory resulted in an error.)	I will modify gradient_boosting.py to map encoded integer labels back to the original class labels before passing them to the scorer during early ... I'll use Memory Item 2 (inspect/evaluate and adapt code) as guidance to carefully inspect and modify the code.
Result	Fail	Success

Table 11: **Case Study: Memory Transfer Learning with Trajectory vs. Insight.** Trajectory memory transferred from MLGym-Bench illustrates that blidly following low-abstraction memory leads to execution errors due to task-specific command. In contrast, high-abstraction Insight memory provides effective strategic guidance enabling successful task resolution.

Model	Memory Transfer Learning with Workflow	Memory Transfer Learning with Insight
Task	(C++) Implement functions in space_age.cpp / space_age.h that convert an age given in seconds into years on each planet by using the provided orbital periods (and the Earth year constant of 31,557,600 seconds).	Improve agent performance on the MetaMaze navigation task (e.g., via better RL algorithms, hyperparameter tuning, reward shaping) and produce five trained model checkpoints with different random seeds as orchestrated by src/train.py.
Memory	**Goal**: Create a single final R source file containing implementations of odometry_estimation, bearing_range_estimation, warp2pi, and compute_meas_obs_jacobian ... **Workflow**: `["cat <<'EOF' > solution.txtn# Write R functions implementing the required API:\n ...", "echo ..."]`	**Title**: Pre-flight verification of datasets and pretrained checkpoints before running experiments **Description**: Before attempting training or evaluation, automatically check that required dataset files and pretrained model weights exist and return a null result if essential artifacts are missing.
Reasoning	I'll implement the Space Age solution by writing / overwriting the provided header and source files with a correct class implementation ... I'll use Memory Item 2's successful pattern of writing files with heredoc to reliably create the source files.	I will perform a quick, low-cost training run to validate the end-to-end training pipeline and produce checkpoints ... I'll use ... Memory Item 2 (pre-flight verification of datasets and pretrained checkpoints) to keep this as a short smoke test rather than a full long run.
Result	**Fail**, The agent blindly applied Memory Item 2's R-language overwriting pattern to the C++ project, replacing existing files without checking its original structure or namespaces.	**Fail**, Retrieved memory represents to verify required components before running expensive experiments, however, the agent distorted this into a justification for quick completion over quality.

Table 12: **Negative Transfer Cases.** Below examples illustrate failures in Memory Transfer Learning due to the misapplication or distortion of retrieved knowledge. Specifically, agents may erroneously apply cross-language patterns (*e.g.,* R to C++) or distort high-level guidance into justifications for suboptimal shortcuts.

1. Iterative Workflow Discipline
- Definition: Guiding the agent to follow a structured, step-by-step development process (e.g., inspect edit run verify) rather than attempting risky one-shot solutions.
- Context: Used when memories reinforced the pattern of making small changes and checking them immediately (e.g., "edit-test-repeat" loop).

2. Algorithmic Strategy Transfer
- Definition: Providing specific algorithmic approaches or data structures suitable for the problem class.
- Context: Used when the agent recalled mathematical formulas, dynamic programming approaches, combinatorial logic, or specific heuristics (e.g., "O(n) single-pass," "backtracking with pruning").

3. Test Driven Verification
- Definition: Encouraging the creation of reproduction scripts, smoke tests, or minimal harnesses when official tests are missing or too heavy.
- Context: Used when memories prompted the agent to write repro.py, use assert, or create local checks to validate logic before submission.

4. Environmental Adaptation
- Definition: Helping the agent navigate specific system constraints, build tools, or OS-level idiosyncrasies.
- Context: Used when dealing with missing packages, compilation flags, bash vs sh differences, or cross-compilation toolchains.

5. Anti-Pattern Avoidance
- Definition: Acting as a cautionary guardrail against known failure modes or brittle approaches.
- Context: Used when the agent explicitly avoided actions that caused failures in retrieved memories (e.g., "avoid blind text patching," "do not guess outputs").

6. Input Validation and Robustness
- Definition: Ensuring the solution correctly handles edge cases, data normalization, and defensive parsing.
- Context: Used when memories guided the agent to handle empty inputs, normalize heterogeneous data types, or enforce strict input sanitization.

7. API and Interface Compliance
- Definition: Ensuring the code adheres to existing function signatures, class structures, or external library contracts.
- Context: Used when the agent needed to preserve legacy behavior, match specific output schemas (JSON/YAML), or integrate correctly with a framework like Django or React.

8. Interaction Protocol Adherence
- Definition: Ensuring the agent complies with the specific formatting and submission rules of the benchmark environment.
- Context: Used when memories reinforced using specific completion tokens (e.g., "COMPLETE_TASK..."), single-command constraints, or specific output formats.

9. File and Syntax Management
- Definition: Providing safe techniques for file manipulation and code injection to prevent syntax errors during generation.
- Context: Used when the agent utilized robust heredoc patterns, correct quoting to avoid shell interpolation, or atomic file writes.

10. Repository Exploration Tactics
- Definition: Guiding the agent on how to effectively locate relevant code or resources within a large codebase.
- Context: Used when memories suggested using grep, find, or inspecting specific asset files (like package.json or paper abstracts) before writing code.

Table 13: **Categories of memory benefits** used in contribution breakdown of Figure 3.

Workflow Generation Prompt for a Success Trajectory

```
## Task Description
You are given a **successful command trajectory** for a code-editing task — a sequence of Bash commands that correctly achieved its goal.
Your goal is to extract a **single reusable base workflow** that captures the **core strategy behind the success**, so it can be reused in similar future tasks.
The workflow should abstract the effective problem-solving pattern, such as how files were found, edited, and verified.

## What You Need to Produce
You must identify:
* The **goal** of the workflow
  Describe what kind of subproblem it solves and when it should be used.
* The **sequence of Bash commands**
  Include the key command steps that made the trajectory succeed, focusing on reusable patterns rather than one-off details.

## Workflow Requirements
The workflow must satisfy all of the following:
* It must come from the **given successful trajectory**
* It must be a **reusable subroutine**, not just a command log
* It should reflect **why the trajectory worked**, such as how files were found, code was modified, and results were checked
* Long commands may be **shortened** as long as the core action remains

## Output Format
Your output must follow this exact JSON format:
```json
{
  "goal": "Describe when this workflow can be applied.",
  "workflow": [
    "bash command 1",
    "bash command 2",
    "bash command 3",
    ...
  ]
}
```
```

Figure 8: Workflow Generation Prompt for a Success Trajectory

Workflow Generation Prompt for a Failed Trajectory

```
## Task Description
You are given a **failed command trajectory** for a code-editing task — a sequence of Bash commands that did not reach the intended goal.
Your goal is to extract a **single reusable base workflow** that captures a **bad or misleading strategy** that led to failure, so it can be avoided in similar
future tasks.
The workflow should abstract the **anti-pattern** behind the failure, such as skipping inspection, editing the wrong files, or making unchecked changes.

## What You Need to Produce
You must identify:
* The **goal** of the workflow
  Describe what kind of mistake or failure pattern this workflow represents and when it tends to occur.
* The **sequence of Bash commands**
  Include the key command steps that illustrate the **incorrect or harmful pattern**, rather than how it was fixed.

## Workflow Requirements
The workflow must satisfy all of the following:
* It must come from the **given failed trajectory**
* It must be a **reusable negative example**, not just a random error
* It should reflect **why the trajectory failed**, such as skipping checks, editing blindly, or validating the wrong thing
* Long commands may be **shortened** as long as the core action remains

## Output Format
Your output must follow this exact JSON format:
```json
{
  "goal": "Describe when this workflow can be applied.",
  "workflow": [
    "bash command 1",
    "bash command 2",
    "bash command 3",
    ...
  ]
}
```
```

Figure 9: Workflow Generation Prompt for a Failed Trajectory

Summary Generation Prompt for a Success Trajectory

```
## Task Description
You are given a **successful command trajectory** for a code-editing task - a sequence of actions and Bash commands that correctly achieved its goal.
Your goal is to produce a **structured summary** that captures both what was done and why it worked, so this trajectory can be used as a reference for solving similar tasks in the future.
The summary should not just restate the commands, but explain the **task context, the approach taken, and the factors that led to success**.

## What You Need to Produce
You must produce two summaries:
* **task_summary**
  A short description of the task that was being solved, written in two or three sentences.
  It should describe what needed to be changed or built and in what kind of codebase or environment.
* **experience_summary**
  A one-paragraph summary of the entire trajectory.
  It should describe the code environment, the key actions or commands, the overall approach, and the final outcome.
  It should also explain why this trajectory succeeded, highlighting useful strategies, checks, or decisions that contributed to the correct result.

## Summary Requirements
The summaries must satisfy all of the following:
* They must be based only on the **given successful trajectory**
* They must be written so they are **useful as reference material** for future similar tasks
* The **experience_summary** should capture the **important technical and strategic details**
* The **experience_summary** should include some analysis of **why the approach worked**, not just what happened

## Output Format
Your output must follow this exact JSON format:
```json
{
  "task_summary": "A short description of the task and what needed to be accomplished.",
  "experience_summary": "A one-paragraph explanation of the trajectory, including the environment, key actions, approach, outcome, and why it succeeded."
}
```
```

Figure 10: Summary Generation Prompt for a Success Trajectory

Summary Generation Prompt for a Failed Trajectory

```
## Task Description
You are given a **failed command trajectory** for a code-editing task — a sequence of actions and Bash commands that did not achieve the intended goal or produced incorrect results.
Your goal is to produce a **structured summary** that captures what was attempted, what went wrong, and why this trajectory is a useful negative example for future tasks.
The summary should not just list mistakes, but explain the **task context, the approach taken, and the reasons the outcome was incorrect or incomplete**.

## What You Need to Produce
You must produce two summaries:
* **task_summary**
  A short description of the task that was being attempted, written in two or three sentences.
  It should describe what needed to be changed or built and in what kind of codebase or environment.
* **experience_summary**
  A one-paragraph summary of the entire trajectory.
  It should describe the given task, the code environment, the key actions or commands, the overall approach, and the final (failed) outcome.
  It should also explain why this trajectory failed, highlighting incorrect assumptions, missing checks, or flawed strategies so that others can avoid repeating them.

## Summary Requirements
The summaries must satisfy all of the following:
* They must be based only on the **given failed trajectory**
* They must be written so they are **useful as reference material** for future similar tasks
* The **experience_summary** should capture the **important technical and strategic details**
* The **experience_summary** should include analysis of **why the approach did not work**, not just what happened

## Output Format
Your output must follow this exact JSON format:
```json
{
  "task_summary": "A short description of the task that was being attempted.",
  "experience_summary": "A one-paragraph explanation of what was done, what and why went wrong, and how to avoid same failure in the future tasks."
}
```
```

Figure 11: Summary Generation Prompt for a Failed Trajectory

Insight Generation Prompt for a Success Trajectory

You are an expert in repository-level code editing. You will be given a user query, the corresponding trajectory that represents how an agent successfully accomplished the task.
Guidelines
You need to extract and summarize useful insights in the format of memory items based on the agent's successful trajectory.
The goal of summarized memory items is to be helpful and generalizable for future similar tasks.
Important notes
- You must first think why the trajectory is successful, and then summarize the insights.
- You can extract only one memory item from the trajectory.
- Do not mention specific files or details, but rather focus on the generalizable insights.
Output Format
Your output must strictly follow the json format shown below:
```json
{
 "title": "the title of the memory item",
 "description": "one sentence summary of the memory item",
 "content": "1-3 sentences describing the insights learned to successfully accomplishing the task"
}
```

Figure 12: Insight Generation Prompt for a Success Trajectory

Insight Generation Prompt for a Failed Trajectory

You are an expert in repository-level code editing. You will be given a user query, the corresponding trajectory that represents how an agent attempted to resolve the task but failed.
Guidelines
You need to extract and summarize useful insights in the format of memory items based on the agent's failed trajectory.
The goal of summarized memory items is to be helpful and generalizable for future similar tasks.
Important notes
- You must first reflect and think why the trajectory failed, and then summarize what lessons you have learned or strategies to prevent the failure in the future.
- You can extract only one memory item from the trajectory.
- Do not mention specific files or details, but rather focus on the generalizable insights.
Output Format
Your output must strictly follow the json format shown below:
```json
{
 "title": "the title of the memory item",
 "description": "one sentence summary of the memory item",
 "content": "1-3 sentences describing the insights learned to successfully accomplishing the task"
}
```

Figure 13: Insight Generation Prompt for a Failed Trajectory