

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

Kangsan Kim¹ Minki Kang¹ Taeil Kim¹ Yanlai Yang² Mengye Ren^{2†} Sung Ju Hwang^{1,3†}

¹KAIST ²New York University ³DeepAuto.ai

<https://memorytransfer.github.io/>

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%, 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

† Equal advising. Correspondence to: kksan07@kaist.ac.kr

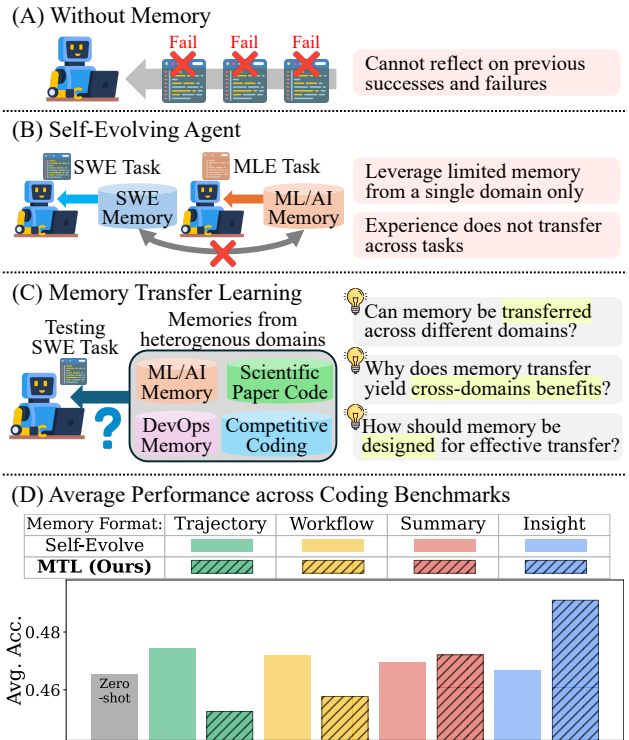


Figure 1. **Conceptual overview of Memory Transfer Learning.** Unlike (A) memory-less agents or (B) single-domain self-evolving agents, (C) our approach utilizes a shared memory pool from heterogeneous coding tasks. (D) In the evaluation on diverse benchmarks, MTL outperforms a self-evolving approach.

in self-evolving agents by enabling the extraction of reusable workflows and transferable insights from past inferences and their application 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 (Yang et al., 2024). Leveraging this knowledge allows agents to reference successful solution patterns in similar tasks, thereby reducing reasoning overhead, while also avoiding unnecessary failure actions in long-horizon code editing through adherence to accumu-

lated procedural and strategic guidance, such as small-step modification heuristics and verification routines.

While memory-augmented coding agents have shown promise (Ouyang 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 applicable to 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 works 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 domains?

RQ3. Which factors in memory transfer learning most influence transfer effectiveness?

To address these open questions, we conduct a systematic investigation of **Memory Transfer Learning** 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), Workflow (Wang et al., 2024c), Summary (Shinn et al., 2023), and Insight (Ouyang et al., 2025), as illustrated in Figure 2. We then evaluate coding agent performance in zero-shot setting and under Memory Transfer Learning. The results demonstrate that Memory Transfer Learning can provide effective and transferable knowledge, **improving 3.7% of average scores of 6 coding benchmarks**.

Our analysis yields three core findings into the mechanism of memory transfer. **First, cross-domain significantly im-**

prove 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 Insights, become task-agnostic and generalizable. In contrast, low-abstraction memories like Trajectories 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 Memory Transfer Learning, 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.

In conclusion, this work presents a first holistic investigation of Memory Transfer Learning. Importantly, through extensive evaluation on six coding benchmarks, we show that existing agents’ memory usage methods, which focus on a homogeneous domain, are limited, and that there is significant room for improvement by leveraging memories from heterogeneous domains. We hope this study expands the scope of memory utilization beyond single-domain settings and stimulates further research on how to effectively leverage memory in self-evolving agents, ultimately leading to more capable coding agents.

2. Related Work

2.1. 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 lan-

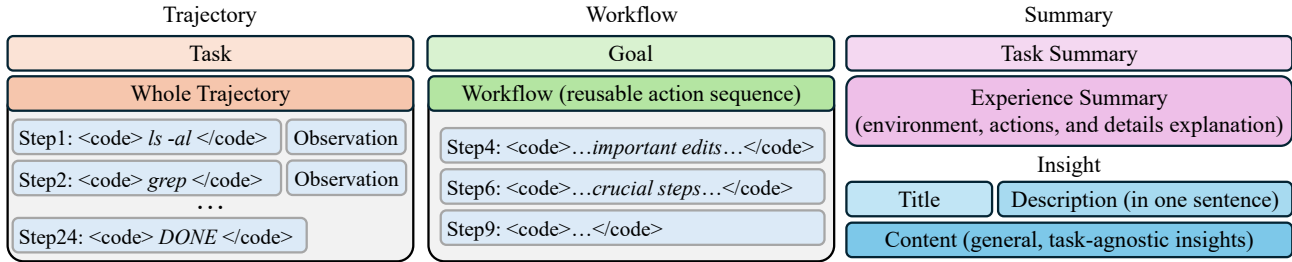


Figure 2. **Illustrative examples of four memory formats.** We utilize Trajectory, Workflow, Summary, and Insight formats to analyze how different levels of information abstraction affect cross-task transferability.

guage 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), RLCoder (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.

2.2. 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 that may be highly beneficial to agent performance.

2.3. 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 & Ruder, 2018; Houlisby 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, it does not provide a deeper analysis of the underlying mechanisms of memory transfer, including which forms of knowledge are transferable and how transfer-oriented memories should be generated in contrast to in-domain knowledge. Moreover, prior work typically constructs unified memory spaces across heterogeneous environments, such as general reasoning, web interaction, and coding, thereby missing the opportunity to exploit coding-specific shared principles that are unique to programming tasks.

3. Memory Transfer Learning

We introduce Memory Transfer Learning, which leverages memories generated from heterogeneous tasks with target tasks in coding environments. In the following sections, we first describe how we generate and retrieve memory, and which benchmarks we use to evaluate the performance.

3.1. 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: memory 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 during the inference.

3.1.1. MEMORY GENERATION

Before memory generation, we first run inference 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 $H = (t, [(r_1, a_1, o_1), \dots, (r_n, a_n, o_n)])$ with task t . 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 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 is as follows. The structure illustration for each format is shown in Figure 2, and prompts used in memory generation are in Appendix E.

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 $M_T = (t, [(a_1, o_1), \dots, (a_n, o_n)])$. This contains detailed information of task solving experience even with failed steps. Also the agent can implicitly estimate the expected execution results of certain actions by referring observations of similar commands in this memory.

Workflow In order to focus only on meaningful code snippets in the entire trajectory, this memory representation is generated by extracting 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 denotes as $M_W = (g, [a_i, a_j, \dots, a_k])$. By restoring a subset of the action and observation history, Workflow is much shorter than Trajectory which leads to less danger of distractions from unrelated 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 provide explicit information about analysis why the agent succeeds or fails and the findings from the history. Thus, for Summary memory, we prompt LLM to summarize the task, environment, actions, results, and analysis on why this inference succeeds or fails from the given trajectory. In detail, LLM generates a summary of task s_t and one paragraph of experience summary s_e from the trajectory, which is represented as $M_S = (s_t, s_e)$ for Summary memory M_S .

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 memory format. Following the memory design ReasoningBank (Ouyang et al., 2025), Insight M_I consist 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 accomplished without mentioning specific files or

details. Additionally, we explicitly instruct LLM to generate generalizable insights for future similar tasks.

3.1.2. MEMORY RETRIEVAL

Memory Pool Construction After finishing memory generation for all benchmarks, we construct the heterogeneous-domain memory pool to experiment memory transfer learning. We gather memories from all benchmarks except the testing benchmark for each memory format. In formal notation, the memory pool \mathcal{P} used for memory transfer learning in evaluating benchmark B_i with memory type τ is $\mathcal{P}_\tau(B_i) = \{M_\tau^{(k)} \mid t^{(k)} \notin B_i\}_{k=1}^{N_i}$. When constructing the memory pools, we index each memory by extracting embedding features using a textual embedding model and store the features with the memories.

Memory Retrieval In the inference stage, we retrieve N relevant memories for each task from the memory pool correspond to the testing model, benchmark, and memory type, and provide 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 retrieved memories by top- N sampling with the highest similarity scores.

3.2. Experimental Details

3.2.1. DATASETS

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

3.2.2. ADDITIONAL DETAILS

We adopt gpt-5-mini model for every LLM usage from generating memories, base model for coding agent, to a LLM judge. 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 the memory retrieval stage,

Table 1. Evaluation results of Memory Transfer Learning. We report Pass@3 scores across multiple benchmarks. MTL consistently improves performance over the zero-shot baseline across models. Among memory types, Insight achieves the highest average performance.

	LiveCodeBenchv6	Aider-Polyglot	SWEBench-Verified	TerminalBench2	ReplicationBench	MLGym-Bench	Avg.
GPT-5-mini							
Zero-shot	0.910	0.470	0.730	0.315	0.111	0.667	0.523
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%
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 2. Pass@3 comparison with self-evolving methods. LCB, SWEB, and RepliB denote LiveCodeBenchv6, SWEBench-Verified, and ReplicationBench, respectively.

Method	#Memories	LCB	SWEB	RepliB	Avg.
Zeroshot	-	0.910	0.730	0.111	0.584
ReasoningBank	97	0.920	0.750	0.133	0.601
AgentKB	5,899	0.920	0.720	0.200	0.613
MTL (Ours)	431	0.930	0.770	0.189	0.630

we select three memories for each query ($N = 3$). 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.

4. Experimental Results and Analysis

We now present and analyze the results of our experiments, and introduce academic findings on characteristics of memory transfer learning based on analysis in diverse aspects.

4.1. Overall Performance of MTL

4.1.1. MAIN RESULTS

The results of the coding agent with Memory Transfer Learning across six coding benchmarks are shown in Table 1. Performance of Memory Transfer Learning significantly improves the performance compared to zero-shot setting, in particular, when Insight memories are transferred, the agent achieves more than 4.0% (up to 8.3%) performance gains on four benchmarks. These results highlight that transferable knowledge exists across different domains, and leveraging such knowledge is crucial for improving the performance of coding agents. Moreover, these results are further vali-

dated across different models. We evaluate the effectiveness of Memory Transfer Learning on the DeepSeek V3.2 (Liu et al., 2025) 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-sourced models, highlighting the broad applicability of cross-domain memory transfer.

4.1.2. COMPARISON WITH SELF-EVOLVING APPROACHES

We further evaluate Memory Transfer Learning against two representative self-evolving methods, ReasoningBank (Ouyang et al., 2025) and AgentKB (Tang et al., 2025), on three benchmarks. Each model is evaluated over three runs, and we report Pass@3 scores to ensure the robust comparison. As presented in Table 2, Memory Transfer Learning outperforms both self-evolving methods by +2.9% and +1.7%, respectively, demonstrating that its performance gain are substantial even relative to strong self-evolving baselines. ReasoningBank achieves the lowest 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, Memory Transfer Learning uses only 431 memories in the memory pool, yet achieves the highest average performance. This demonstrates both the effectiveness and efficiency of our approach compared to existing self-evolving methods.

Core Finding 1. Memory Transfer Learning significantly improves coding agent performance and outperforms self-evolving methods in effectiveness and efficiency.

Table 3. Case Study: Zero-shot vs. Memory Transfer Learning with Insight. Transferred Insight from LiveCodeBench 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.

Model	Zero-shot	Memory Transfer Learning with Insight
Memory	(No Memory)	Memory Item 2 (Generated from LiveCodeBench): 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

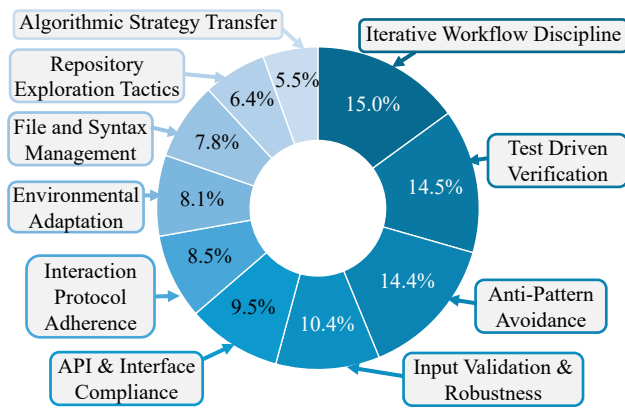


Figure 3. Breakdown of Memory Transfer Contribution. Transferred memory mainly contributes through meta-knowledge.

4.2. Mechanism of Memory Transfer Learning

4.2.1. HOW DOES MEMORY TRANSFER LEARNING BENEFIT THE AGENTS?

To investigate the operational mechanisms of memory transfer learning, we inspect the inference outcomes using LLM and manual case studies. First, we collect the trajectories of the instances in which the agent fails in the zero-shot setting but succeeds when Memory Transfer Learning with Insight memory is applied, and use GPT-5 to categorize how transferred memory contributes to successful task completion. As presented in Figure 3, our analysis reveals that transferred memory primarily benefits agents by providing meta-knowledge rather than task-specific programming 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. By supplying procedural guidance on how to act and how to safely interact with the execution and testing environment, Memory Transfer Learning enables agents to follow stable inference patterns and significantly reduces failure cases caused by infrastructure-level errors. On the other hand, *Algorithmic Strategy Transfer* accounts for only 5.5% of the total gains in Figure 3, suggesting that the direct transfer of specific programming knowledge or algorithms is limited in our setting.

4.2.2. CASE STUDY: ZERO-SHOT VS. MTL

The effect of Memory Transfer Learning and transferred meta-memory are also shown in the case study. In Table 3, we compare the inference outcomes between zero-shot and memory transfer learning tested on one instance of SWEBench-Verified. In zero-shot setting, model naively solve the task by simply raising an error and eventually fail the test. However, with memory transfer learning, retrieved Insight memory generated from LiveCodeBench 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 case study highlights the practical impact of transferred meta-memory in enabling successful task 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.

4.3. Impact of Memory Abstraction

4.3.1. 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-

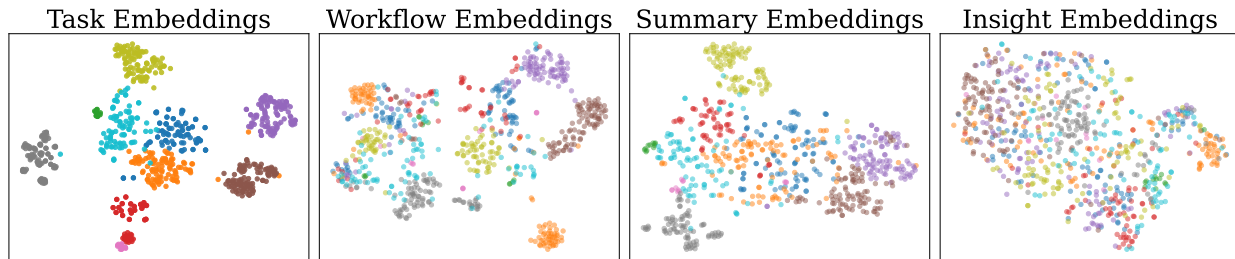


Figure 4. **t-SNE Visualization of Memory Formats.** The leftmost plot shows task embeddings, followed by three different memory types to the right. Each color represents a specific benchmark used in experiments. While task and workflow embeddings are clustered within each domain, the insight embeddings are sparse and intermingled, reflecting their task-agnostic nature.

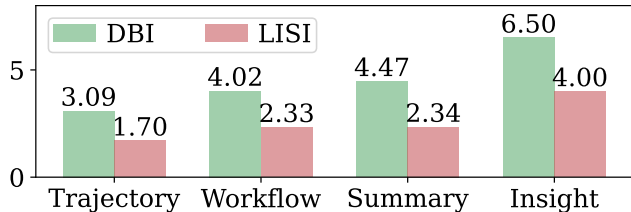


Figure 5. **Embedding Distribution Analysis** DBI and LISI reveal weaker separation and stronger mixing with higher abstraction.

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, quantitatively supporting the transition from task-specific to generalized memory.

4.3.2. CORRELATION BETWEEN ABSTRACTION AND TRANSFER EFFECTIVENESS

In Table 1, under the memory transfer learning 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 4.2.1, transferred memories primarily provide general meta-knowledge, whereas implementation-specific details from unrelated tasks may distract the agent. Consequently, more abstract and generalized memory representations tend to yield higher transfer effectiveness.

4.3.3. ISOLATING THE EFFECT OF MEMORY ABSTRACTION

To isolate the effect of abstraction, we compare two groups of memories within the same representation (Insight): rel-

Table 4. **Memory Abstraction Effect** Task-agnostic insights outperform task-specific ones, highlighting abstraction as a key driver.

Method	LCB	SWEB	RepliB	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%

atively 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). This controlled setup allows us to evaluate the effect of abstraction while keeping the memory format fixed. In Table 4, we observe that even within the same memory format, task-agnostic memories consistently outperform task-specific ones. This provides evidence that abstraction, rather than format itself, is a key factor driving transfer performance. Furthermore, we present the formal modeling on the correlation of memory abstraction and transfer effectiveness in Appendix C.

4.3.4. CASE STUDY: TRAJECTORY VS. INSIGHT

We further validate the relationship between memory abstraction level and transferability through qualitative case studies. In Table 5, we compare representative examples of memory transfer using Trajectory and Insight format. In the Trajectory transfer example, the agent blindly follows the exact commands-level instructions in the memory. This behavior is inherently risky, since implementation details often differ across tasks and environments, including programming languages, file structures, and execution pipelines. As a result, the transferred Trajectory memory acts as a brittle anchor, leading the agent to execute incompatible commands and ultimately causing runtime errors.

In contrast, the transferred Insight memory provides high-level behavioral guidance, such as prioritizing inspection of evaluation criteria and improving data utilization by merging

Table 5. Case Study: Memory Transfer Learning with Trajectory vs. Insight. Trajectory memory transferred from MLGym-Bench illustrates that blindly 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 Trajectory	Memory Transfer Learning with Insight
Memory	<p>Memory Item 1 (Generated from MLGym-Bench):</p> <p>Task: You are given the train, validation and test sets of Housing Price Prediction dataset and a baseline...</p> <p>Trajectory:</p> <pre> 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 .drwxrwxr- 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' ... </pre>	<p>Memory Item 1 (Generated from MLGym-Bench):</p> <p>Title: Inspect eval requirements, combine train+val, use robust preprocessing, and adapt to library API changes</p> <p>Description: Before improving a baseline model, read the repository/evaluation expectations, train on all available labeled data, apply robust preprocessing for numeric and categorical ...</p> <p>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 ...</p>
Reasoning	<p>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.)</p>	<p>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.</p>
Result	Fail	Success

training and validation sets. 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 reflected in the reasoning trace, the agent internalizes these general coding practices while deriving task-specific implementation details, leading to successful task completion.

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

4.4. Further Analysis and Ablations

4.4.1. NEGATIVE TRANSFER IN MTL

As shown in Table 1, Memory Transfer Learning can degrade performance in certain benchmarks. To understand this, we analyzed instances where zero-shot setting succeeded but Memory Transfer Learning 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 instead of formal criteria, resulting in missed specifications and silent failures.
- **Misapplied best-practice transfer:** Successful patterns are sometimes transferred indiscriminately, overriding task-specific semantics. This causes procedural over-engineering and rigid adherence to familiar workflows that violate new task requirements.

We find that major three reasons of negative transfer are caused by wrong memory retrieval and failed adaptation of the retrieved memory to the new task. These demonstrate that we can avoid performance degradation by designing advanced memory retrieval methods that retrieve truly helpful memories not semantically relevant items, and employ better memory adaptation methods, such as memory rewriting module (Cao et al., 2025).

4.4.2. CASE STUDY: NEGATIVE MEMORY 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 Appendix B, 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 misinterpreted as a justification for suboptimal shortcuts.

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

4.4.3. IMPACT OF THE MEMORY POOL SIZE

To investigate how Memory Transfer Learning scales with the number of memory in the candidate pool, we evaluate Memory Transfer Learning with varying memory pool sizes across three benchmarks. Specifically, we randomly sample memories from the full cross-domain memory pool at

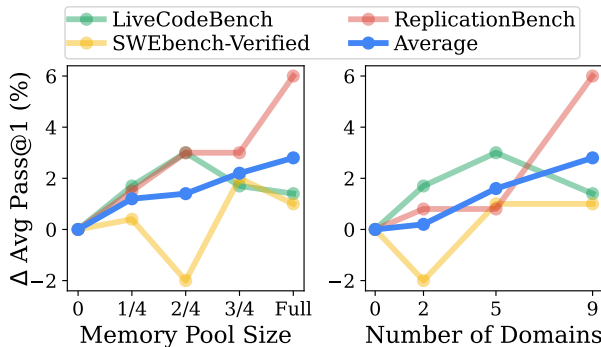


Figure 6. **Memory Scaling** Larger memory pools and more domains lead to better performance through increased diversity.

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, indicating that larger memory pools lead to better performance. This trend arises because a larger pool increases the likelihood of retrieving relevant memories for the target task.

Furthermore, we evaluate our method using varying numbers of memory source domains (benchmarks) to examine how performance scales. We find that the average performance gain generally increases as the number of source domains grows. In particular, using 9 domains yields the best overall performance. These results demonstrate that the effectiveness of Memory Transfer Learning benefits from incorporating a larger number of domains. This trend suggests that a broader set of domains enhances the diversity of transferable knowledge, thereby increasing the likelihood of retrieving useful meta-knowledge for target tasks.

Finding 5. The effectiveness of Memory Transfer Learning scales with the size of the memory pool and the number of domains.

4.4.4. CROSS-MODEL MEMORY TRANSFER LEARNING

To validate whether memories are transferable across models, we evaluate agent performance under Memory Transfer Learning using memories generated by different models. We hypothesize that if Memory Transfer Learning 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. The results, shown in Table 6, consistently outperform the zeroshot baseline even when using memories from other models. In particular, cross-model memory transfer is effective in both directions, from a stronger model (GPT-5-mini) to weaker models (Qwen3-Coder and DeepSeek V3.2), and vice versa. These findings support our hypothesis that meta-knowledge is transferable across models because it is model-agnostic. However, cross-model transfer consistently underperforms

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

Source	Target	LCB	SWEB	RepliB	Avg.
Zeroshot	GPT-5-mini	0.863	0.623	0.059	0.515
DeepSeek V3.2	GPT-5-mini	0.890	0.617	0.048	0.518
Qwen3-Coder	GPT-5-mini	0.883	0.607	0.093	<u>0.528</u>
GPT-5-mini	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	DeepSeek V3.2	0.890	0.450	0.163	<u>0.501</u>
DeepSeek V3.2	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	Qwen3-Coder	0.780	0.347	0.111	0.413
Qwen3-Coder	Qwen3-Coder	0.740	0.370	0.130	0.413

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

Method	LCB	SWEB	RepliB	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	<u>0.608</u>
Embedding Similarity	0.930	0.770	0.189	0.630

compared to MTL using self-generated memories. This suggests that model-specific biases may exist in the memories.

Finding 6. Memory can be transferred across different models, while self-generated memories yield the best performance.

4.4.5. ANALYSIS ON RETRIEVAL METHODS

As discussed in Section 4.4.1, 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).

Finding 7. Cross-domain memory retrieval is inherently challenging, and static retrieval methods fail to generalize in heterogeneous agentic settings.

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 single-domain settings and stimulates further research into robust memory usage strategies for self-evolving coding agents.

Impact Statement

This paper presents work whose goal is to advance the field of self-evolving coding agents, specifically by introducing Memory Transfer Learning to leverage knowledge across heterogeneous domains. By enabling agents to effectively transfer high-level meta-knowledge, our work contributes to making agentic systems more generalizable and data-efficient, reducing the need for extensive domain-specific fine-tuning. This has positive implications for lowering 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.

References

- Cai, Z., Guo, X., Pei, Y., Feng, J., Su, J., Chen, J., Zhang, Y.-Q., Ma, W.-Y., Wang, M., and Zhou, H. Flex: Continuous agent evolution via forward learning from experience. *arXiv preprint arXiv:2511.06449*, 2025.
- Cao, R., Chen, M., Chen, J., Cui, Z., Feng, Y., Hui, B., Jing, Y., Li, K., Li, M., Lin, J., et al. Qwen3-coder-next technical report. *arXiv preprint arXiv:2603.00729*, 2026.
- Cao, Z., Deng, J., Yu, L., Zhou, W., Liu, Z., Ding, B., and Zhao, H. Remember me, refine me: A dynamic procedural memory framework for experience-driven agent evolution. *arXiv preprint arXiv:2512.10696*, 2025.
- Chen, S., Lin, S., Gu, X., Shi, Y., Lian, H., Yun, L., Chen, D., Sun, W., Cao, L., and Wang, Q. Swe-exp: Experience-driven software issue resolution. *arXiv preprint arXiv:2507.23361*, 2025.
- Chou, J., Liu, A., Deng, Y., Zeng, Z., Zhang, T., Zhu, H., Cai, J., Mao, Y., Zhang, C., Tan, L., Xu, Z., Zhai, B., Liu, H., Zhu, S., Zhou, W., and Lian, F. Autocodebench: Large language models are automatic code benchmark generators, 2025. URL <https://arxiv.org/abs/2508.09101>.
- Dong, Q., Li, L., Dai, D., Zheng, C., Ma, J., Li, R., Xia, H., Xu, J., Wu, Z., Chang, B., Sun, X., Li, L., and Sui, Z. A survey on in-context learning. In Al-Onaizan, Y., Bansal, M., and Chen, Y.-N. (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 1107–1128, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.64. URL <https://aclanthology.org/2024.emnlp-main.64/>.
- Fang, J., Peng, Y., Zhang, X., Wang, Y., Yi, X., Zhang, G., Xu, Y., Wu, B., Liu, S., Li, Z., et al. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*, 2025a.
- Fang, R., Liang, Y., Wang, X., Wu, J., Qiao, S., Xie, P., Huang, F., Chen, H., and Zhang, N. Memp: Exploring agent procedural memory. *arXiv preprint arXiv:2508.06433*, 2025b.
- Gao, H.-a., Geng, J., Hua, W., Hu, M., Juan, X., Liu, H., Liu, S., Qiu, J., Qi, X., Wu, Y., et al. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.
- Gauthier, P. Aider polyglot benchmark. Blog post and benchmark details, 2024. URL <https://aider.chat/2024/12/21/polyglot.html>.
- Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., Attariyan, M., and Gelly, S. Parameter-efficient transfer learning for nlp. In *International conference on machine learning*, pp. 2790–2799. PMLR, 2019.
- Howard, J. and Ruder, S. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.
- Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

- Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.
- Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=VTF8yNQM66>.
- Kim, K., Park, G., Lee, Y., Yeo, W., and Hwang, S. J. Videoicl: Confidence-based iterative in-context learning for out-of-distribution video understanding. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 3295–3305, 2025.
- Kim, K., Yang, Y., Kim, S., Yeo, W., Lee, Y., Ren, M., and Hwang, S. J. Ma-egoqa: Question answering over egocentric videos from multiple embodied agents. *arXiv preprint arXiv:2603.09827*, 2026.
- Liu, A., Mei, A., Lin, B., Xue, B., Wang, B., Xu, B., Wu, B., Zhang, B., Lin, C., Dong, C., et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.
- Luo, Q., Ye, Y., Liang, S., Zhang, Z., Qin, Y., Lu, Y., Wu, Y., Cong, X., Lin, Y., Zhang, Y., et al. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*, 2024.
- Merrill, M. A., Shaw, A. G., Carlini, N., Li, B., Raj, H., Bercovich, I., Shi, L., Shin, J. Y., Walshe, T., Buchanan, E. K., et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.
- Min, S., Lyu, X., Holtzman, A., Artetxe, M., Lewis, M., Hajishirzi, H., and Zettlemoyer, L. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*, 2022.
- Mitchener, L., Laurent, J. M., Andonian, A., Tenmann, B., Narayanan, S., Wellawatte, G. P., White, A., Sani, L., and Rodrigues, S. G. Bixbench: a comprehensive benchmark for llm-based agents in computational biology. *arXiv preprint arXiv:2503.00096*, 2025.
- Nathani, D., Madaan, L., Roberts, N., Bashlykov, N., Menon, A., Moens, V., Budhiraja, A., Magka, D., Vorotilov, V., Chaurasia, G., et al. Mlgym: A new framework and benchmark for advancing ai research agents. *arXiv preprint arXiv:2502.14499*, 2025.
- Ouyang, S., Yan, J., Hsu, I., Chen, Y., Jiang, K., Wang, Z., Han, R., Le, L. T., Daruki, S., Tang, X., et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140*, 2025.
- Ridnik, T., Kredo, D., and Friedman, I. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.
- Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- Seo, M., Baek, J., Lee, S., and Hwang, S. J. Paper2code: Automating code generation from scientific papers in machine learning. *arXiv preprint arXiv:2504.17192*, 2025.
- Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.
- Suzgun, M., Yuksekgonul, M., Bianchi, F., Jurafsky, D., and Zou, J. Dynamic cheatsheet: Test-time learning with adaptive memory. *arXiv preprint arXiv:2504.07952*, 2025.
- Tang, X., Qin, T., Peng, T., Zhou, Z., Shao, D., Du, T., Wei, X., Xia, P., Wu, F., Zhu, H., Zhang, G., Liu, J., Wang, X., Hong, S., Wu, C., Cheng, H., Wang, C., and Zhou, W. Agent KB: leveraging cross-domain experience for agentic problem solving. *arXiv preprint arXiv:2507.06229*, 2025.
- Team, H. F. Harbor framework: A framework for evaluating and optimizing agents and models in container environments. <https://github.com/laude-institute/harbor>, 2026.
- Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024a.
- Wang, Y., Wang, Y., Guo, D., Chen, J., Zhang, R., Ma, Y., and Zheng, Z. Rlcoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487*, 2024b.
- Wang, Z. Z., Mao, J., Fried, D., and Neubig, G. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024c.
- Xia, C. S., Deng, Y., and Zhang, L. Top leaderboard ranking = top coding proficiency, always? evoeval: Evolving coding benchmarks via llm. *arXiv preprint*, 2024.

- Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K. R., and Press, O. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL <https://arxiv.org/abs/2405.15793>.
- Ye, C., Yuan, S., Cooray, S., Dillmann, S., Roque, I. L., Baron, D., Frank, P., Martin-Alvarez, S., Koblichke, N., Qu, F. J., et al. Replicationbench: Can ai agents replicate astrophysics research papers? *arXiv preprint arXiv:2510.24591*, 2025.
- Yeo, W., Kim, K., Jeong, S., Baek, J., and Hwang, S. J. Universalrag: Retrieval-augmented generation over corpora of diverse modalities and granularities. *arXiv preprint arXiv:2504.20734*, 2025a.
- Yeo, W., Kim, K., Yoon, J., and Hwang, S. J. Worldmm: Dynamic multimodal memory agent for long video reasoning. *arXiv preprint arXiv:2512.02425*, 2025b.
- Zhang, G., Ren, H., Zhan, C., Zhou, Z., Wang, J., Zhu, H., Zhou, W., and Yan, S. Memevolve: Meta-evolution of agent memory systems. *arXiv preprint arXiv:2512.18746*, 2025.
- Zhang, K., Li, J., Li, G., Shi, X., and Jin, Z. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*, 2024.
- Zheng, L., Wang, R., Wang, X., and An, B. 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, 2024. URL <https://openreview.net/forum?id=Pc8AU1aF5e>.
- Zhong, L., Wang, Z., and Shang, J. Debug like a human: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*, 2024.
- Zhu, Q., Guo, D., Shao, Z., Yang, D., Wang, P., Xu, R., Wu, Y., Li, Y., Gao, H., Ma, S., et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.
- Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., and He, Q. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76, 2020.

A. Average Pass@1 Results

Table 8. Evaluation results of Memory Transfer Learning.

	LiveCodeBench	Aider-Polyglot	SWEBench-Verified	TerminalBench2	ReplicationBench	MLGym-Bench	Avg.
GPT-5-mini							
ZeroShot	0.863	0.343	0.623	0.206	0.059	0.583	0.435
MTL (T)	0.890	0.357	0.610	0.195	0.063	0.528	0.438
MTL (W)	0.877	0.350	0.620	0.243	0.081	0.583	0.449
MTL (S)	0.887	0.370	0.613	0.228	0.078	0.611	0.451
MTL (I)	0.877	0.347	0.633	0.213	0.119	0.639	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	0.357	0.347	0.210	0.126	0.500	0.366
MTL (I)	0.740	0.360	0.370	0.228	0.130	0.528	0.377
Δ	+0.7%	+0.3%	+2.3%	+1.9%	+0.4%	+2.8%	+1.2%
DeepSeek V3.2							
ZeroShot	0.890	0.433	0.423	0.285	0.144	0.500	0.446
MTL (I)	0.893	0.447	0.463	0.288	0.178	0.556	0.466
Δ	+0.3%	+1.3%	+4.0%	+0.4%	+3.3%	+5.6%	+2.0%

B. Case Study on Negative Transfer

Table 9. **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.

Model	Memory Transfer Learning with Workflow	Memory Transfer Learning with Insight
Task	(C++) Implement functions in <code>space_age.cpp / space_age.h</code> 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 <code>src/train.py</code> .
Memory	Goal: Create a single final R source file containing implementations of <code>odometry_estimation</code> , <code>bearing_range_estimation</code> , <code>warp2pi</code> , and <code>compute_meas_obs_jacobian</code> ... Workflow: <code>["cat <<'EOF' > solution.txt\n # Write R functions implementing the required API:\n ...", "echo ..."]</code>	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.

C. Formal Modeling of Abstraction

To formally ground these 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 Tradeoff). Under these assumptions, our formal model proves that the expected empirical transfer gain strictly increases with the abstraction level A .

D. Memory Benefit Category

In [Table 10](#), we present the categories of memory contributions generated by the LLM for analysis.

Table 10. Categories of Memory Benefits

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.
-

E. Memory Generation Prompts

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 7. 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 8. 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 9. 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 10. 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 11. 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 12. Insight Generation Prompt for a Failed Trajectory