Demystify the Role of Memory in Machine Learning Engineering Agents

Xinyu Zhao[1*], **Junpeng Wang**[2], **Yuzhong Chen**[2], **Menghai Pan**[2],
Chin-Chia Michael Yeh[2], **Jiarui Sun**[2], **Yan Zheng**[2],
Mahashweta Das [2], **Tianlong Chen**[1]
[1]University of North Carolina at Chapel Hill, [2]Visa Research

Abstract

While memory is a core component in agent systems, its behavioral impact in complex, long-horizon domains like machine learning engineering (MLE) remains poorly understood. Unlike short, reactive exchanges, MLE agents solve tasks through cycles of experimentation and improvement where past errors can inform future success. This paper presents a systematic study dissecting how memory influences agent behavior and performance across diverse MLE challenges. We first introduce a dynamic coding memory designed to capture and reuse debugging experiences, and integrate it into two representative agent paradigms: a **sequential, chain-based agent** that mirrors human-like iterative refinement, and a **parallel, tree-based agent** that performs broad, self-exploratory search in the code space. Our central finding is that the role of memory is contingent on the agent's underlying architecture. For chain-based agents, memory proves highly beneficial, enabling them to **avoid recurring mistakes and engage in more coherent, iterative refinement**, which significantly improves reliability and task success. In contrast, for tree-based search agents, memory introduces a critical trade-off: it **enhances procedural stability at the cost of constraining search diversity**, which can prematurely narrow exploration and lead to suboptimal final solutions. These findings reveal a fundamental trade-off between procedural reliability and solution innovation modulated by memory, offering insights for designing more effective and robust MLE agents.

1 Introduction

As artificial intelligence (AI) evolves, it is increasingly tasked with solving complex, open-ended problems that define the "second half" of its development, such as systematic scientific discovery (Yao et al., 2023). An example of such a challenge is machine learning engineering (MLE), where the goal is to automate and optimize the entire data science workflow (Huang et al., 2024; Jing et al., 2024; Chan et al., 2025; Hong et al., 2024a). This has led to the development of agents that leverage large language models (LLMs) to autonomously handle the full machine learning lifecycle, from understanding tasks and datasets to engineering features, selecting models, and evaluating results (Sun et al., 2024; Zheng et al., 2025; Lu et al., 2025).

To tackle the complexity of real-world MLE tasks, two prominent agent paradigms have emerged. The first is the **solution-evolution framework**, exemplified by AIDE (AI-Driven Exploration) (Jiang et al., 2025), which frames MLE as a code optimization problem and structures the trial-and-error process as a tree search over possible solutions. The second is the **sequential, chain-based paradigm**, represented by agents like OpenHands (Wang et al., 2024), which mimics a human developer's workflow through an iterative, linear loop of actions and observations. While these approaches have achieved state-of-the-art performance on benchmarks like MLE-Bench (Chan et al., 2025), our analysis reveals that, despite their architectural differences, both are fundamentally constrained by repetitive and inefficient behaviors. As illustrated in Figure 1, agents frequently get stuck in error loops, either by repeating the same mistake within a single exploratory tree or by making identical errors across independent runs of the same task. This core challenge stems from several shared issues: (1) **Structural Isolation**: The agent fails to transfer knowledge effectively, both within a single solution tree and across different tasks, leading to redundant exploration. (2) **Inefficient Processes**: The reliance on simple greedy policies or linear reasoning often traps agents in local optima or repetitive error-correction cycles (Toledo et al., 2025; Liu et al., 2025).

Recent advances underscore the power of mem-

[*]Work was done during the internship at Visa Research.

Findings of the Association for Computational Linguistics: ACL 2026, pages 10811–10826
July 2-7, 2026 ©2026 Association for Computational Linguistics

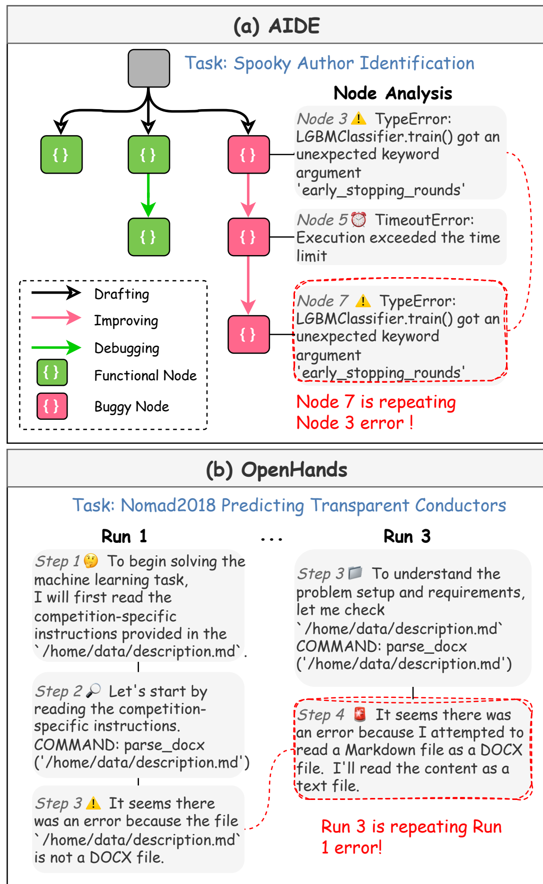

Figure 1: Illustration of repetitive error patterns in MLE agents. (a) In AIDE, the agent repeats an error that had previously occurred during exploration. (b) In OpenHands, the agent reproduces the same file-format error across different runs for the same task.

ory as an essential enhancement for LLM agents, enabling both long-horizon reasoning and the accumulation of knowledge across interactions (Zhang et al., 2024b; Chhikara et al., 2025; Li et al., 2023; Zhang et al., 2024a; Lu et al., 2023; Packer et al., 2023). However, these memory systems are predominantly crafted for conversational contexts, such as dialogue history tracking, session recall, or user preferences, rather than for the structured, code-centric workflows of machine-learning exploration. This paper presents a systematic empirical study investigating how memory modulates MLE agent behavior across different architectural paradigms. To enable this investigation, we introduce Dynamic Coding Memory, a structured mechanism designed to mitigate repetitive errors by allowing agents to retain, retrieve, and update experiences from previous executions. Unlike general-purpose conversational memory, our approach is built for code-centric reasoning and organizes knowledge at the granularity of both tasks and coding packages. We integrate our memory module into the two representative agent architectures: the parallel, tree-based AIDE and the sequential, chain-based OpenHands, and evaluate their performance and behavior on the comprehensive MLE-Bench benchmark. The key contributions of this paper are:

- We design and implement Dynamic Coding Memory, a task-aware memory system specifically tailored for the structured, code-centric workflows of machine learning exploration.
- We present a core empirical finding that memory acts as a double-edged sword: it consistently improves the reliability and success of sequential agents by preventing recurring errors, but it can hinder parallel search agents by reducing exploratory diversity, leading to safer but ultimately suboptimal solutions.
- We provide a detailed quantitative analysis of how memory rebalances the trade-off between exploration and exploitation. We show that for parallel search agents, memory strongly favors exploitation—promoting safer, local refinements (evidenced by a 22.7% bug rate reduction) at the cost of exploration (measured by reduced search diversity). For sequential agents, it encourages a more effective exploitation, leading to more coherent refinement. We also note that these behavioral shifts are further nuanced by data modality.

2 Related Works

LLM Agents for Machine Learning Engineering Recent work leverages large language models (LLMs) to build agentic workflows across diverse domains, such as software engineering (Liu et al., 2024; Hong et al., 2024b; Chen et al., 2023; Islam et al., 2024; Anthropic, 2025; Google, 2025), web automation (Zhou et al., 2023; Abuelsaad et al., 2024; Lu et al., 2025), and embodied settings (Li et al., 2024; Guo et al., 2024b). A particularly promising frontier is LLM agents for scientific discovery (Gridach et al., 2025), focusing on the end-to-end automation of machine learning engineering (MLE) (Sun et al., 2024; Jing et al., 2024; Zhang et al., 2025a). MLE that traditionally requires extensive manual effort for data pre-processing, modeling, and evaluation, is well-suited for agent-based automation. Two primary paradigms have emerged for structuring these MLE

agents. The first is chain-based reasoning, where agents refine solutions through stepwise reasoning-and-acting loops, easing the interpretation of intermediate logic. For example, DS-Agent employs a Reflexion-inspired (Shinn et al., 2023) debugger that analyzes execution feedback to reflect on potential bugs before generating corrected code (Guo et al., 2024a). Other iterative agents combine external tools and knowledge as complementary context. MLE-STAR uses a search engine to retrieve effective models from the web to form an initial solution, which it then iteratively refines (Nam et al., 2025). The second paradigm is exploratory search, which enables agents to consider multiple potential solution paths simultaneously. This approach frames problem-solving as a tree or graph search, allowing for more robust and comprehensive exploration (Yao et al., 2023). A prominent example is AIDE (Jiang et al., 2025), which treats MLE tasks as code optimization problems solvable via a structured tree search. While this paradigm is powerful for navigating complex solution spaces, it introduces significant challenges in managing the search process efficiently. Despite architectural differences, existing MLE agents largely treat memory as a uniformly beneficial component, without systematically examining how it reshapes exploration dynamics under different agent paradigms.

Agent Memory In the context of LLM agents, memory is the process of acquiring, storing, retaining, and subsequently retrieving information to inform future actions (Wu et al., 2025; Zhang et al., 2025b; Xiong et al., 2025). It is a critical component for enabling the long-term reasoning and knowledge accumulation required to solve complex, real-world problems. Recognizing its importance, many general-purpose memory architectures have been proposed, from systems enhancing the personalization of conversational agents (OpenAI, 2024; Xu et al., 2025; Packer et al., 2023) to frameworks designed for long-term task completion and tool use (Zhang et al., 2024a; Li et al., 2023). In LLM agents for MLE, initial works have highlighted the importance of memory. MLZero, a multi-agent system for automating the ML pipeline, employs semantic and episodic memory to iteratively refine code (Fang et al., 2025). ML-Master utilizes a selectively scoped memory mechanism to share insights and successful code snippets between sibling nodes in its exploration process (Liu et al., 2025). While these systems prove memory is beneficial, they rely on a flat storage of past interactions. Another gap exists for a task-aware memory framework that organizes knowledge structurally, generalizing insights at a higher semantic level (*e.g.*, by code libraries) to prevent repetitive errors and promote knowledge reuse across tasks. Meanwhile, recent work has studied the role of diversity in MLE agents. Zhang et al. (2026) propose mechanisms to boost solution diversity through learned ideation, and AlphaEvolve (Novikov et al., 2025) maintains a database of programs to encourage varied exploration. These works focus on promoting diversity, while our study provides the systematic empirical evidence that memory has a concrete impact on MLE agents' diversity, and quantify how it reshapes the exploration-exploitation balance.

3 Methodology

3.1 Preliminary

We study machine-learning engineering (MLE) agents aiming to produce runnable programs. For a task t with dataset $D_t = (\mathcal{X}_t, \mathcal{Y}_t)$ and an evaluation functional $\mathrm{Eval}_t : \mathcal{Z} \times D_t \to \mathbb{R}$, a candidate implementation is $z = (f, \lambda) \in \mathcal{Z}$, where f is a runnable Python program and λ is a light configuration (*e.g.*, random seed, flags). Executing z yields a scalar validation score

$$J_t(z) \;=\; \mathrm{Eval}_t\big(z; D_t\big). \tag{1}$$

The objective is to identify a high-scoring implementation:

$$z^\star \in \arg\max_{z \in \mathcal{Z}} \; J_t(z). \tag{2}$$

Rather than formulating MLE as a long-horizon control problem, we evaluate fully specified programs independently.

3.2 Representative Agent Frameworks

We instantiate two representative MLE agent architectures that stand for different paradigms of machine learning exploration: a *chain-based* event–observation loop and a *tree-based* code-space search. Both serve as foundations for integrating memory in MLE agents.

Chain-based MLE Agent OpenHands (Wang et al., 2024) is a generalist coding agent that interacts with a sandboxed OS through a small set of programming-language–native actions (run bash, run Python/IPython, and browse). The agent state

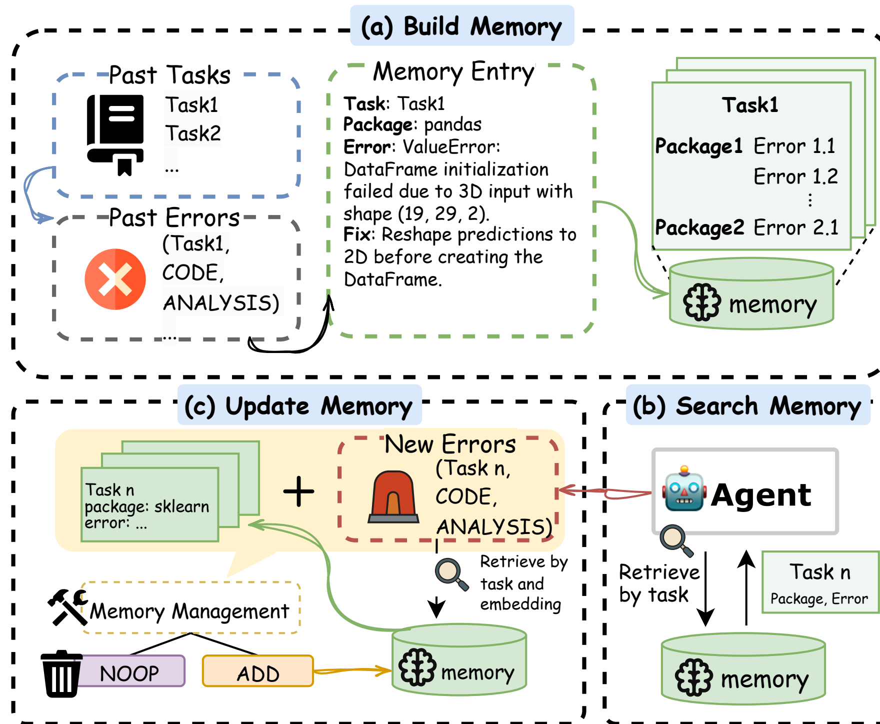

Figure 2: Overview of the proposed dynamic coding memory. (a) Build Memory: past tasks and error–fix pairs are encoded into structured memory entries, organized by task and package. (b) Search Memory: during each new run, the agent retrieves relevant experiences based on task similarity and embedding-level relevance to guide current decisions. (c) Update Memory: new errors and fixes are integrated online, with a lightweight management module that filters duplicates or outdated entries (NOOP/ADD). This dynamic mechanism enables continuous learning across runs while reducing repetitive coding errors.

is represented by an event stream, a chronological log of actions and their corresponding observations. At each iteration, the agent applies an API

$$\texttt{step(state)} \rightarrow \texttt{Action},$$

The agent's primitive operations include:

- EXECUTE: run Python or shell commands to test or evaluate code,
- OBSERVE: read console outputs or error traces returned from the environment,
- REVISE: modify or extend code snippets based on observed outcomes.

Each interaction thus forms a step in a sequential reasoning loop that mirrors human debugging and refinement behavior. We use OpenHands representing chain-based experiments, enabling the memory module to provide contextual reminders and prior error knowledge during the iterative updates.

Tree-based MLE Agent (code-space search). **AIDE** (Jiang et al., 2025) frames the MLE process as AI-Driven Exploration in the space of executable code. AIDE maintains a dynamic solution tree where each node corresponds to a runnable Python script with an associated validation score. A search policy Π alternates among operations:

- DRAFT: generate a new candidate solution from scratch as a root of a new branch;
- DEBUG: repair a previously failed node while preserving its design intent;
- IMPROVE: modify a valid script incrementally to enhance its performance metric.

Each script is executed to obtain its validation score $J_t(z)$ (Eq. 2), and a summarization operator condenses relevant historical information such as code edits, scores, and errors to form concise prompts.

3.3 Dynamic Coding Memory

Large language model–based machine learning exploration (MLE) agents often exhibit repetitive error patterns when attempting to generate or optimize runnable code. Such repetition stems from recurring root causes including software package incompatibilities, misuse of application programming interfaces, or mismatched data schema definitions. These issues frequently reappear not only

within a single task branch but also across independent runs and related competitions.

To mitigate this inefficiency, we propose a Dynamic Coding Memory mechanism that allows agents to retain, retrieve, and update structured experiences extracted from previous executions. Unlike general conversational memory designed for open-domain dialogue, this module is built for code-centric reasoning and is organized at the granularity of both task and package, enabling fine-grained transfer of debugging knowledge across semantically similar problem contexts.

Memory Construction and Representation. During each task execution, whenever the generated code fails to execute successfully, the system parses the error traceback and forms a structured record of the failure event. Each memory entry is represented as

$$e = \big(t,\ \text{package},\ \text{error},\ \text{fix},\ \phi(e)\big) \tag{3}$$

where t denotes the task identifier, package indicates the software module associated with the failure (*e.g.*, `lightgbm`, `sklearn`), error stores a canonicalized version of the runtime traceback, fix describes the corresponding remedy that successfully resolves the error, and $\phi(e) \in \mathbb{R}^d$ is an embedding computed from the textual concatenation of the error and its fix summary. To improve the generalization of the memory representation, we use "general" as a package to cover those errors which could not be simply attributed to a specific package or module, such as I/O-related issues. All entries are saved into a task–package–structured memory library (Figure 2a).

Memory Retrieval and Update. Unlike general chatting memory, the current query in coding tasks does not reveal the to-be-written solution. Moreover, general memory retrieved through RAG often provides limited complementary signals for code-specific failures. Therefore, we adopt a task-indexed retrieval strategy: for the current task t, the agent only searches memory entries recorded under the same task. Formally, we first restrict the library to the task-specific subset:

$$\mathcal{M}_t \;=\; \big\{\, e \in \mathcal{M} \;\big|\; \text{task}(e) = t \,\big\}.$$

As the agent continues exploration, new experiences are continuously added to the memory system. Whenever a new execution yields an error, a temporary record $e' = (t,\ \text{package}',\ \text{error}',\ \text{fix}', \phi(e'))$ is produced through LLM and a text embedding model. Before adding this record, the system checks whether it is redundant with respect to existing memories from the same task and package. Specifically, it retrieves the subset of entries in $\mathcal{M}$ that share the same task identifier t and package name, and ranks them by embedding similarity between representation $\phi(\cdot)$. We set the number of existing similar entries to compare to 5. If the new error-fix pair is highly similar to any stored entry, it is skipped. Otherwise, the record is appended to the memory library and its embedding vector is stored for future retrieval. This lightweight redundancy check ensures that the memory grows only when novel or substantially improved debugging knowledge is encountered, keeping $\mathcal{M}$ compact, diverse, and focused on non-overlapping experiences.

Integration with Agentic Search. The dynamic coding memory operates as a modular component compatible with both agent paradigms described in Section 3.2. In the *chain-based agent* (OpenHands), retrieved entries are used to construct contextual prompts that help the agent prevent repeating known package-specific mistakes and suggest verified code revisions in the initial message for each attempt to propose a solution. In the *tree-based agent* (AIDE), memory guides both the DEBUG and IMPROVE operations by warning the agent of previously resolved errors when generating the next solution in the search tree. In both cases, the memory system transforms short-term reactive corrections into persistent, transferable knowledge, leading to higher success rates and more stable exploration behaviors across runs and tasks.

4 Experiments and Analyses

4.1 Experimental Setup

Implementation Details We select AIDE and OpenHands as they represent the two fundamental paradigms—tree-based search and sequential chain—that underpin most existing MLE agents. For example, DS-Agent (Guo et al., 2024a) and MLE-STAR (Nam et al., 2025) follow the chain-based pattern, while ML-Master (Liu et al., 2025) and MLZero (Fang et al., 2025) build on tree-based exploration. We use o3 (OpenAI, 2025) as the backbone LLM for AIDE, as studies show reasoning models bring much gain over the tree-search framework (Jiang et al., 2025; Chan et al., 2025). We deploy GPT-4o as the backbone LLM of OpenHands,

Table 1: Results from Scaffolding and Models experiments. Experiments on AIDE are repeated with 16 seeds, while those for OpenHands are repeated with 3 seeds. Scores represent the mean $\pm$ one standard error of the mean.

Model	Made Submission (%)	Valid Submission (%)	Above Median (%)	Bronze (%)	Silver (%)	Gold (%)	Any Medal (%)
AIDE (o3)							
Baseline	100.00±0.00	91.27±0.83	**55.47±1.94**	**14.76±1.45**	9.03±1.81	**10.61±1.61**	**34.40±1.83**
+ Memory	100.00±0.00	**94.80±0.95**	44.77±1.61	7.72±1.35	**10.36±0.92**	4.87±0.77	22.95±1.77
OpenHands (gpt-4o)							
Baseline	31.82±6.94	28.79±6.06	15.15±4.01	13.64±2.62	0.00±0.00	1.52±1.52	15.15±4.01
+ Memory	**34.85±1.52**	**34.85±1.52**	**19.70±1.52**	**16.67±1.52**	0.00±0.00	1.52±1.52	**18.18±0.00**
MLAB (gpt-4o) [Estimated/Rebuttal]							
Baseline	36.36±4.50	22.72±3.80	4.54±2.00	4.54±2.00	0.00±0.00	0.00±0.00	4.54±2.00
+ Memory	**40.90±4.20**	**31.81±3.50**	4.54±2.00	4.54±2.00	0.00±0.00	0.00±0.00	4.54±2.00

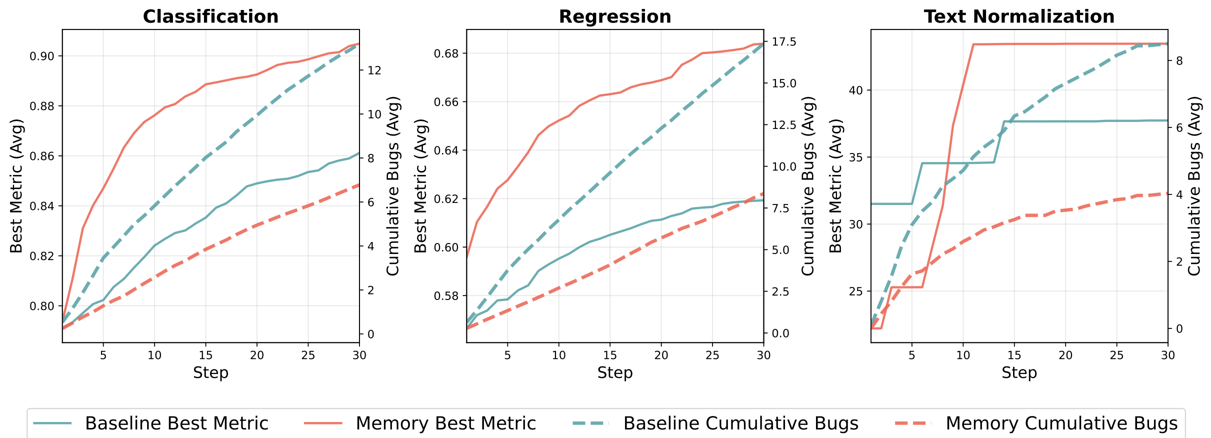

Figure 3: Averages of best-so-far validation metric and cumulative numbers of bugs for AIDE Baseline and Memory over all steps. For regression task group, metrics are transformed as $1/(1 + loss)$ (higher better).

following the setting in MLE-Bench (Chan et al., 2025). More implementation details are listed in Appendix 6. All experiments are conducted on a compute server running Ubuntu 20.04, equipped with dual AMD EPYC 7513 32-core processors (128 vCPUs), 448 GB RAM, and a single NVIDIA A100-SXM4 GPU (40 GB VRAM). The system provides 2 TB of SSD storage with full read–write access for intermediate results and final outputs. Each LLM agent is allocated a 24-hour wall-clock budget per task. For AIDE, all experiments are repeated 16 times per task and 3 times for OpenHands. We report the mean and standard error of the mean across runs.

Datasets MLE-bench (Chan et al., 2025) comprises 75 tasks sourced from Kaggle, designed to assess the capabilities of agents in machine learning engineering. In our experiments, we evaluate on MLE-bench lite, a curated subset of 22 tasks selected from the full benchmark. We compare each agent's submission with those of human participants on the official leaderboard of each Kaggle competition, and average across all designated splits. We adopt the MLE-Bench evaluation metrics, consisting of two categories: (1) Competition Ranking Metrics. Agents are evaluated on simulated Kaggle leaderboards following its official medal system. Each submission can earn a bronze, silver, or gold medal based on its percentile among all teams, with thresholds adjusted by competition size to ensure consistent difficulty. The headline metric is the proportion of attempts that win any medal (bronze or above), providing a single competitiveness indicator comparable to top human Kagglers. We also include Above-Median Ranking, denoting whether a submission ranks higher than the median team. (2) Submission-related metrics. They capture the agent's ability to complete and

validate submissions, including Made Submission, which indicates whether a valid file was produced, and Valid Submission, which measures the ratio of successfully evaluated submissions to all attempts.

4.2 Main Results and Observations

Table 1 presents the aggregated outcomes of integrating our proposed dynamic coding memory into two representative MLE agent frameworks: the tree-based AIDE and the chain-based OpenHands. The introduction of memory yields mixed yet informative effects. Across both paradigms, memory integration yields measurable improvements in stability, though with distinct behavioral implications.

In OpenHands, which executes tasks through a single linear reasoning loop, the primary bottleneck lies in error recovery during debugging. The incorporation of memory substantially alleviates this limitation. The Made and Valid Submission rates increase by $+3.0\%$ and $+6.1\%$ points, respectively, accompanied by a $+3.0$ point improvement in Any-Medal achievement. These results confirm that memory enables the chain-based agent to avoid recurring implementation mistakes and stabilize iterative code refinement, directly addressing the dominant failure mode in sequential workflows.

In contrast, AIDE operates as a self-exploration system that concurrently expands and revises multiple branches within a solution tree. While memory enhances the Valid Submission Rate by $+3.53\%$ points, it leads to notable declines in Above-Median (-10.7%) and Any-Medal (-11.5%) outcomes. This degradation suggests that when exploration already occurs at scale, memory may introduce an unintended bias toward previously validated but suboptimal solution regions, reducing the diversity of candidate branches. Consequently, memory acts as a stabilizer in sequential agents but a regularizer in exploratory agents, balancing efficiency and risk at the cost of innovation. These contrasting effects motivate our subsequent behavioral analyses on (1) Early Errors vs. Final Generalization, (2) Exploration vs. Exploitation, and (3) Task-Type Sensitivity, to further dissect how memory modulates learning dynamics across agent architectures.

4.3 Memory-Induced Reliability vs. Generalization in Self-Exploring

To examine how memory influences the balance between exploration safety and final performance, we conduct a step-level analysis of AIDE, as it implements autonomous self-exploration. Specifically, we compare the best-so-far validation metric and cumulative bug number across search steps averaged over all runs. We aggregate results from tasks sharing metrics (classification and ranking, regression-loss-like, and text-normalization; full task lists and per-task objectives are in Appendix 8). For regression-loss-based objectives, we map the raw loss to a higher-is-better surrogate $1/(1+loss)$. As shown in Figure 3, the memory-augmented AIDE achieves a substantial 23% reduction in cumulative bug frequency across all task groups, demonstrating markedly improved execution reliability during search. However, this increased safety does not consistently translate to superior final outcomes. As shown in the aggregated results, the improvement in best-achieved metric varies by task family—classification and regression tasks benefit from more stable progression curves, while text normalization tasks plateau early. Across 22 tasks, 11 exhibit performance gains whereas 10 show slight declines, resulting in a weak Pearson correlation coefficient ($r = 0.21$) between bug reduction and final validation performance.

These findings reveal a behavioral trade-off: memory promotes procedural stability, minimizing redundant or premature failures, but may also reduce search diversity by discouraging exploratory risk-taking. Especially in self-exploring agents like AIDE, excessive emphasis on prior error avoidance can constrain the discovery of novel, high-reward configurations. This insight motivates our next analysis on the Exploration–Exploitation Dynamics, where we quantitatively assess how memory reshapes the structural diversity of code trajectories within the solution tree.

4.4 Memory-Rebalanced Search: Exploration vs. Exploitation

To quantify how agent memory reshapes search behavior, we evaluate two complementary axes: (1) exploration breadth, captured by the diversity of tools/packages and code-level operators, and (2) exploitation intensity, captured by the degree of local refinement around a viable solution.

For AIDE, exploration breadth is computed on runnable nodes by estimating the Shannon entropy H of (i) imported software packages and (ii) AST-level operators, and reporting the effective number of categories $N_{\text{eff}} = exp(H)$ (i.e., the size of an equally frequent partition with the same entropy). Exploitation is assessed via (a)

Table 2: Aggregated exploration–exploitation metrics for AIDE, averaged across all tasks and runs. Each entry reports the mean ± SEM, with Δ denoting the paired difference (Memory–Baseline).

	Baseline	+Memory	Δ
Bug rate (%)↓	46.72±1.58	24.02±1.06	**-22.7**
N_{eff}↑	8.45±0.16	8.10±0.14	**-0.34**
First valid node↓	3.12±0.30	0.80±0.14	**-2.32**
Valid code dist.↓	29.17±0.54	27.18±0.41	**-0.02**

Table 3: Aggregated exploration–exploitation metrics for OpenHands, averaged across paired runs. Values indicate mean ± SEM, and Δ denotes Memory–Baseline.

	Baseline	+Memory	Δ
Bug rate (%)↓	16.27±1.48	14.11±1.66	**-2.16**
N_{eff}(tool)↑	2.13±0.08	1.93±0.09	**-0.20**
N_{eff}(command)↑	115.24±23.60	181.19±28.19	**+65.95**
N_{eff}(ops)↑	0.66±0.17	0.53±0.16	**-0.13**

time-to-first-valid node (smaller indicates earlier commitment to a workable approach) and (b) the distance between consecutive valid codes using token-set Jaccard distance, with a smaller value indicating tighter, more iterative refinement). For OpenHands, exploration breadth is measured at two levels: (i) tool diversity, defined over high-level action types (e.g., `IPythonRunCellAction`, `ShellCommandAction`, `FileEditAction`), and (ii) command diversity, defined over the issued command/invocation strings within a tool family. In addition, we also compute package and AST-operator diversity on successful steps and report $N_{\text{eff}} = exp(H)$.

From the results in Table 2, memory substantially reduces the overall bug rate (-22.7%) and accelerates the steps it takes to the first runnable solution (-2.32%). These improvements coincide with a slight contraction in exploration breadth (-0.34%) and a marginally smaller distance between valid codes (-0.02%). The pattern suggests that memory regularizes AIDE's self-exploration, suppressing repeated failure modes and promoting locally consistent refinements at the expense of broad search diversity. For OpenHands, memory yields a moderate decline in bug rate (-2.16%). Tool diversity decreases slightly (-0.20%), yet command diversity expands markedly ($+65.95\%$). Together, these outcomes indicate that memory enhances OpenHands stability and promotes more coherent, iterative refinement, while reducing redundant failures with constructive exploitation rather than extensive exploration.

4.5 Modality Sensitivity of Memory

To further understand how memory influences performance across different types of machine learning tasks, we analyze per-task and per-modality outcomes. Figure 4 visualizes the relative improvement in "Any Medal" rate for each task, while Table 5 summarizes the aggregated changes by data modality. Overall, memory reduces error rates consistently across modalities, confirming its role in enhancing reliability. However, its impact on final performance varies by task type. For AIDE, the largest gains appear in tabular tasks ($+16.25\%$), where structured reasoning benefits most from past correction, while image and audio tasks exhibit declines (-23.13% and -5.00%, respectively). In contrast, OpenHands shows more balanced improvements, with notable gains in text ($+11.11\%$) and tabular ($+8.33\%$) domains, reflecting the advantage of linguistic and symbolic memory reuse in single-trajectory exploration.

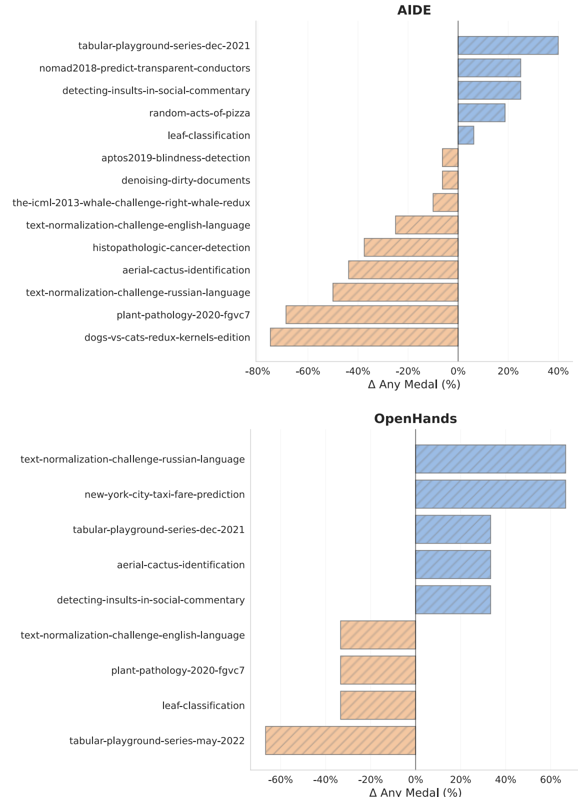

Figure 4: Per-task improvement in "Any Medal" rate for memory-augmented agents compared to baselines.

4.6 When Memory Helps and When It Hurts

The previous analyses characterize memory's effects in aggregate. Here we trace its mechanism through two representative AIDE tasks. Table 4

Table 4: Representative memory entries accumulated by AIDE during exploration. Each entry records the originating task, the responsible package, the error encountered, and the fix that resolved it.

Task	Package	Error	Fix
leaf-classif.	lightgbm	`TypeError: LGBMClassifier.fit()` got unexpected keyword argument `early_stopping_rounds`	Use callback-based early stopping: `callbacks=[lgb.early_stopping(50)]`
leaf-classif.	xgboost	`TypeError: XGBClassifier.fit()` got unexpected keyword argument `eval_metric`	Move `eval_metric` to the `XGBClassifier(...)` constructor
aerial-cactus	timm	`LocalEntryNotFoundError`: attempted to download pretrained weights without internet access	Set `pretrained=False` in `timm.create_model`
denoising-dirty	sklearn	`TypeError: HistGradientBoostingRegressor` does not accept `n_jobs`	Remove `n_jobs=-1` from initialization

Table 5: Per-modality performance comparison between baseline and memory-augmented MLE agents. Δ denotes absolute and relative change in mean $\pm$ SEM.

Modality	Baseline	Memory	Δ
AIDE			
Audio	25.00 ± 17.68	20.00 ± 14.14	-5.00 ↓
Image	37.50±12.09	14.37±7.24	-23.13 ↓
Tabular	18.75±16.24	35.00±20.46	16.25 ↑
Text	35.42± 15.21	30.21±11.06	-5.21 ↓
OpenHands			
Audio	0.00 ± 0.00	0.00 ± 0.00	0.0 →
Image	13.33 ± 9.66	10.00±6.75	-3.33 ↓
Tabular	25.00 ± 13.82	33.33±16.67	8.33 ↑
Text	16.67 ± 6.80	27.78 ± 12.21	11.11 ↑

lists concrete memory entries accumulated during exploration.

Success: `leaf-classification`. Without memory, the agent encounters persistent API-version mismatches—`TypeErrors` from deprecated keyword arguments in `lightgbm` and `xgboost` (Table 4, rows 1–2)—that recur across branches. With memory, actionable fixes are stored early and retrieved frequently (74% of buggy nodes). The bug rate drops from 91.7% (440/480 nodes) to 21.2% (102/480), with the dominant `TypeError` falling from 260 to 18 occurrences. This demonstrates memory's core strength: converting ephemeral debugging outcomes into reusable knowledge that transfers across branches of the search tree.

Failure: `dog-breed-identification`. Memory reduces the bug rate (39.8%→34.6%), yet the best-so-far score degrades substantially (0.622→0.294). A four-step causal chain explains this: **(1)** Of 1,348 stored entries, 548 (40.7%) are timeout errors reflecting execution limits rather than code bugs. **(2)** On buggy nodes, 55/95 retrievals (57.9%) return timeout guidance, steering the agent toward cheaper approaches. **(3)** Deep image packages decline on successful steps (`torch`: 280→107; `timm`: 20→3) while tabular stacks rise (`lightgbm`: 1→35; `catboost`: 0→26; `sklearn`: 289→314). **(4)** The agent converges on lightweight models ill-suited for this image task. Timeout memories act as an implicit regularizer that penalizes compute-intensive but task-appropriate strategies—a concrete instance of the diversity-reduction mechanism identified in Section 4.3. Additional per-run trajectory traces for these and other tasks are provided in Appendix C.

5 Conclusion

In this work, we conducted a systematic investigation into the impact of memory on machine learning engineering agents. This study highlights an inherent trade-off between execution reliability and exploratory innovation in memory-augmented MLE agents. Our findings also connect to the broader study of diversity in MLE agents, providing the first empirical evidence that memory is a concrete source of diversity reduction, and quantifying how this effect varies across agent architectures. Future work could explore adaptive memory mechanisms that dynamically balance exploitation and exploration, for instance by down-weighting memory's influence during early-stage broad exploration and strengthening it during later-stage debugging. Our case studies further reveal that memory quality matters as much as memory quantity. In sum, understanding the role of memory in MLE agents is essential for building the next generation of intelligent agents capable of robustly and creatively solving complex real-world problems.

Limitations

Our study primarily investigates how memory shapes the behavior and performance of MLE agents. While we identify several characteristic behavioral patterns, our analyses are limited to existing benchmark environments and predefined agent architectures. Future research should evaluate whether these findings generalize to real-world machine learning engineering workflows with more dynamic data, resource, and feedback conditions. Our current memory design uses task-indexed retrieval, which does not transfer knowledge across different tasks. A cross-task retrieval ablation shows that intra-task similarity is 2–4.9× higher than cross-task similarity (t=10.69, p<1e-9), suggesting that cross-task transfer is a promising but methodologically challenging direction, as task ordering would introduce an additional confound. In addition, because memory enables persistent behavioral patterns and reminds agents of past errors, it also introduces potential risks of bias amplification and error propagation, where agents may inadvertently reinforce flawed reasoning from earlier attempts. Understanding and mitigating these pathological memory effects will be essential for developing more reliable and interpretable MLE agents. Adaptive memory scheduling, e.g., disabling memory during early exploration to preserve search diversity, is another concrete direction worth investigating, particularly for tree-based agents.

References

Tamer Abuelsaad, Deepak Akkil, Prasenjit Dey, Ashish Jagmohan, Aditya Vempaty, and Ravi Kokku. 2024. Agent-e: From autonomous web navigation to foundational design principles in agentic systems. *arXiv preprint arXiv:2407.13032*.

Anthropic. 2025. Claude code.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. 2025. Mle-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F Karlsson, Jie Fu, and Yemin Shi. 2023. AutoAgents: A framework for automatic agent generation. *arXiv preprint arXiv:2309.17288*.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. 2025. Mem0: Building production-ready AI agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*.

Haoyang Fang, Boran Han, Nick Erickson, Xiyuan Zhang, Su Zhou, Anirudh Dagar, Jiani Zhang, Ali Caner Turkmen, Cuixiong Hu, Huzefa Rangwala, and 1 others. 2025. MLZero: A multi-agent system for end-to-end machine learning automation. *arXiv preprint arXiv:2505.13941*.

Google. 2025. Open code.

Mourad Gridach, Jay Nanavati, Khaldoun Zine El Abidine, Lenon Mendes, and Christina Mack. 2025. Agentic AI for scientific discovery: A survey of progress, challenges, and future directions. *arXiv preprint arXiv:2503.08979*.

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. 2024a. DS-Agent: Automated data science by empowering large language models with case-based reasoning. *arXiv preprint arXiv:2402.17453*.

Xudong Guo, Kaixuan Huang, Jiale Liu, Wenhui Fan, Natalia Vélez, Qingyun Wu, Huazheng Wang, Thomas L Griffiths, and Mengdi Wang. 2024b. Embodied LLM agents learn to cooperate in organized teams. *arXiv preprint arXiv:2403.12482*.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Li Zhang, Lingyao Zhang, Min Yang, Mingchen Zhuge, Taicheng Guo, Tuo Zhou, Wei Tao, Wenyi Wang, Xiangru Tang, Xiangtao Lu, and 6 others. 2024a. Data interpreter: An LLM agent for data science. *CoRR*, abs/2402.18679.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, and 1 others. 2024b. Metagpt: Meta programming for a multi-agent collaborative framework. International Conference on Learning Representations, ICLR.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2024. Mlagentbench: Evaluating language agents on machine learning experimentation. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. 2025. AIDE: ai-driven exploration in the space of code. *CoRR*, abs/2502.13138.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. 2024. Dsbench: How far are data science agents to becoming data science experts? *CoRR*, abs/2409.07703.

Manling Li, Shiyu Zhao, Qineng Wang, Kangrui Wang, Yu Zhou, Sanjana Srivastava, Cem Gokmen, Tony Lee, Erran Li Li, Ruohan Zhang, and 1 others. 2024. Embodied agent interface: Benchmarking LLMs for embodied decision making. *Advances in Neural Information Processing Systems*, 37:100428–100534.

Yuan Li, Yixuan Zhang, and Lichao Sun. 2023. MetaAgents: Simulating interactions of human behaviors for LLM-based task-oriented coordination via collaborative generative agents. *arXiv preprint arXiv:2310.06500*.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*.

Zexi Liu, Yuzhu Cai, Xinyu Zhu, Yujie Zheng, Runkun Chen, Ying Wen, Yanfeng Wang, Siheng Chen, and 1 others. 2025. ML-Master: Towards AI-for-AI via integration of exploration and reasoning. *arXiv preprint arXiv:2506.16499*.

Junru Lu, Siyu An, Mingbao Lin, Gabriele Pergola, Yulan He, Di Yin, Xing Sun, and Yunsheng Wu. 2023. MemoChat: Tuning LLMs to use memos for consistent long-range open-domain conversation. *arXiv preprint arXiv:2308.08239*.

Zhengxi Lu, Yuxiang Chai, Yaxuan Guo, Xi Yin, Liang Liu, Hao Wang, Han Xiao, Shuai Ren, Guanjing Xiong, and Hongsheng Li. 2025. UI-R1: Enhancing efficient action prediction of GUI agents by reinforcement learning. *arXiv preprint arXiv:2503.21620*.

Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Sercan Ö Arık, and Tomas Pfister. 2025. MLE-STAR: Machine learning engineering agent via search and targeted refinement. *arXiv preprint arXiv:2506.15692*.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, and 1 others. 2025. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*.

OpenAI. 2024. Memory and new controls for ChatGPT.

OpenAI. 2025. Introducing OpenAI o3 and o4-mini.

Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. 2023. MemGPT: Towards LLMs as operating systems.

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.

Maojun Sun, Ruijian Han, Binyan Jiang, Houduo Qi, Defeng Sun, Yancheng Yuan, and Jian Huang. 2024. A survey on large language model–based agents for statistics and data science. *arXiv preprint arXiv:2412.14222*.

Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Rishi Hazra, Nicolas Baldwin, Alexis Audran-Reiss, Michael Kuchnik, Despoina Magka, Minqi Jiang, Alisia Maria Lupidi, and 1 others. 2025. AI research agents for machine learning: Search, exploration, and generalization in MLE-bench. *arXiv preprint arXiv:2507.02554*.

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

Yaxiong Wu, Sheng Liang, Chen Zhang, Yichao Wang, Yongyue Zhang, Huifeng Guo, Ruiming Tang, and Yong Liu. 2025. From human memory to AI memory: A survey on memory mechanisms in the era of LLMs. *arXiv preprint arXiv:2504.15965*.

Zidi Xiong, Yuping Lin, Wenya Xie, Pengfei He, Jiliang Tang, Himabindu Lakkaraju, and Zhen Xiang. 2025. How memory management impacts LLM agents: An empirical study of experience-following behavior. *arXiv preprint arXiv:2505.16067*.

Wujiang Xu, Kai Mei, Hang Gao, Juntao Tan, Zujie Liang, and Yongfeng Zhang. 2025. A-Mem: Agentic memory for LLM agents. *arXiv preprint arXiv:2502.12110*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Dan Zhang, Sining Zhoubian, Min Cai, Fengzu Li, Lekang Yang, Wei Wang, Tianjiao Dong, Ziniu Hu, Jie Tang, and Yisong Yue. 2025a. DataSciBench: An LLM agent benchmark for data science. *arXiv preprint arXiv:2502.13897*.

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

Yunxiang Zhang, Edan Toledo, Martin Josifoski, Karen Hambardzumyan, Alisia Maria Lupidi, Alexis Audran-Reiss, Michael Kuchnik, Despoina Magka, Minqi Jiang, Nicolas Baldwin, and 1 others. 2026. Learning to ideate for machine learning engineering agents. *arXiv preprint arXiv:2601.17596*.

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. 2024b. A survey on the memory mechanism of

large language model based agents. *ACM Transactions on Information Systems*.

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. 2025b. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6):1–47.

Yuxiang Zheng, Dayuan Fu, Xiangkun Hu, Xiaojie Cai, Lyumanshan Ye, Pengrui Lu, and Pengfei Liu. 2025. DeepResearcher: Scaling deep research via reinforcement learning in real-world environments. *arXiv preprint arXiv:2504.03160*.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, and 1 others. 2023. WebArena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*.

A Implementation details

Table 6 details the hyperparameters for the two tested scaffolds: AIDE (Jiang et al., 2025) and OpenHands (Wang et al., 2024).

Table 6: Scaffold hyperparameters.

AIDE	
Parameter	**Value**
agent.code.model	o3
agent.feedback.model	gpt-4.1
agent.steps	30
agent.search.max_debug_depth	3
agent.search.debug_prob	0.5
exec.timeout	7200
OpenHands	
Parameter	**Value**
agent	CodeActAgent
model	gpt-4o
max_time_in_hours	24
max_steps	500

Table 7: Error-type recurrence: fraction of runs where the same error type appears in both early (steps 0–9) and late (step ≥ 10) phases. Only tasks with non-zero recurrence are shown.

Task	Base	Mem
dog-breed-identification	0.00	0.33
histopathologic-cancer-detection	0.33	0.00
new-york-city-taxi-fare-prediction	0.33	0.33
text-normalization-english	0.33	0.00
text-normalization-russian	0.00	0.33
the-icml-2013-whale-challenge	0.00	0.67

B Memory Retrieval and Error Analysis

Cross-task retrieval ablation. Intra-task retrieval yields a mean embedding similarity of 0.032 ± 0.009, compared to 0.013 ± 0.003 for cross-task retrieval—a 2.0–4.9× gap (t=10.69, p<1e-9, Cohen's d=2.28). The strongest cross-task matches are generic library-level fixes (*e.g.*, API deprecation warnings), whereas intra-task retrieval preserves task-local context. Cross-task retrieval would also introduce task-ordering as an additional confound.

Early-phase error analysis. We analyze how early-phase errors (steps 0–9) relate to error prevention in later steps (step ≥ 10). Table 9 shows that memory reduces late-phase errors on 11/22 tasks, with the largest reductions on tasks where early errors encode critical environmental knowledge (*e.g.*, data schema, package availability). Table 7 further shows that memory eliminates early-to-late error recurrence on several tasks, confirming that early-phase memories break recurring error patterns.

C Annotated Trajectory Examples

Example 1: KeyError cascade .(`siim-isic-melanoma-classification`).
In the baseline, Run 2 falls into a `KeyError: 'sex'` loop at step 13 persisting for 84/263 steps (32%). With memory, all runs exhibit zero `KeyError` occurrences.

Task: siim-isic-melanoma-classif.

```
Baseline Run 1 | 35 steps | late_err=2
  [step  0] PackageNotFoundError
  [step 18] ModuleNotFoundError: 'sklearn'
  [step 21] NameError: 'train_data_encoded'

Baseline Run 2 | 263 steps | late_err=84
  [step  0] NameError: 'parse_markdown'
  [step 13] KeyError: 'sex'
  [step 15] KeyError: 'sex'
  [step 18] KeyError: 'sex'
  ... +245 more KeyError occurrences

Baseline Run 3 | 22 steps | late_err=5
  [step  0] NameError: 'parse_markdown'
  [step 13] ModuleNotFoundError: 'tensorflow'
  [step 16] ModuleNotFoundError: 'tensorflow'
  [step 18] ModuleNotFoundError: 'tensorflow'
  [step 20] ModuleNotFoundError: 'tensorflow'
  [step 21] UnicodeDecodeError

Memory Run 1 | 14 steps  | late_err=0
  [step  0] NameError: 'parse_markdown'

Memory Run 2 | 500 steps | late_err=0
  (no errors)

Memory Run 3 | 25 steps  | late_err=1
  [step  0] NameError: 'parse_markdown'
  [step  9] ModuleNotFoundError: 'sklearn'
  [step 12] TypeError: OneHotEncoder.__init__()
```

Task: histopathologic-cancer-detect.

```
Baseline Run 1 | 346 steps | late_err=0
  [step  0] PackageNotFoundError

Baseline Run 2 | 23 steps | late_err=3
  [step  0] NameError: 'parse_markdown'
  [step  4] ImportError: 'pandas or numpy'
  [step  6] ImportError: 'PIL'
  [step 12] ModuleNotFoundError: 'sklearn'
  [step 18] ModuleNotFoundError: 'tensorflow'
  [step 22] UnicodeDecodeError

Baseline Run 3 | 22 steps | late_err=2
  [step  0] PackageNotFoundError
  [step  9] ModuleNotFoundError: 'sklearn'
  [step 15] ModuleNotFoundError: 'tensorflow'
  [step 21] UnicodeDecodeError

Memory Run 1 | 500 steps | late_err=0
  (no errors)

Memory Run 2 | 5 steps   | late_err=0
  [step  0] NameError: 'parse_markdown'
  [step  4] UnicodeDecodeError

Memory Run 3 | 11 steps  | late_err=1
  [step  0] NameError: 'parse_markdown'
  [step 10] UnicodeDecodeError
```

Figure 5: The prompt from memory summarization.

Memory Summarization Instructions

```
You are an expert in ML coding challenges who reviews errors from previous
    coding attempts and creates concise error summaries to inform future
    coding decisions.

## Task
Extract the **package** and **summary** for each error analysis provided.

## Package Guidelines
- Identify the root cause package responsible for the error
- Look at the input code and analysis to determine the true source
- Use `general` for errors related to:
  - Uninstalled packages
  - External dependencies (data files, network resources)
  - Environment/system issues

## Summary Requirements
Write a concise 1-2 sentence summary containing:
1. **Error description**: Error type and the operation/class/function that
    caused it
2. **Suggested fix**: If mentioned in the analysis (exclude fixes requiring
    new packages or internet access)

## Output Format
```
[PACKAGE] package_name
[ERROR] Error description. Suggested fix if available.
```

## Constraints
- Only include information from the current error analysis
- Do not suggest fixes requiring new package installations
- Do not suggest fixes requiring internet access
- Keep summaries concise and actionable

### Examples
**Input code:** ......

**Output summary:** [PACKAGE] lightgbm
[ERROR] TimeoutError: model training interrupted during fold 4 due to large
    dataset and high n_estimators=500. Reduce n_estimators, add early stopping
    , or optimize data processing to speed up training.
```

Figure 6: The prompt from memory update.

Memory Update Instructions

```
You are a memory manager for ML coding errors. Your job is to decide if a new
    error is the SAME as existing errors.

RULES:
- ADD: If the new error is DIFFERENT from all existing errors
- NONE: ONLY if the new error is EXACTLY the same as an existing error

CRITICAL: Be VERY CONSERVATIVE. If you're not 100% sure they're identical,
    choose ADD.

Decision criteria:
1. Different function names = ADD (e.g., LGBMClassifier.fit() vs lgb.train())
2. Different error types = ADD (e.g., TypeError vs ValueError vs RuntimeError)
3. Different error messages = ADD (even if same package and function)
4. Different packages = ADD (e.g., torch vs torchvision vs lightgbm)
5. Only EXACT same error message = NONE

1. **ADD**
If the retrieved fact contains new information not already present in memory,
    add it by generating a new ID in the id field.

2. **NONE** (No Change)
ONLY if the retrieved fact describes the EXACT same issue (i.e., same package,
     same function/method, same error type, and same root cause), make no
    change.

IMPORTANT: Different functions, different error types, or different root
    causes should be considered as ADD, even if they seem related.

Examples of what should be ADDED (different issues):
- LGBMClassifier.fit() vs lgb.train() (different functions)
- TypeError vs ValueError (different error types)
- BCEWithLogitsLoss vs AttributeError (different error types)
- SyntaxError vs NameError (different error types)
- Different packages (torch vs torchvision vs lightgbm)
- Different error messages even if same package and function

Examples of what should be NONE (same issue):
- Exact same error message
- Same function, same error type, same root cause

EXAMPLES: ......
```

Table 8: MLE-Bench Lite Task Categories.

Task Name	Modality	Metric Group
aerial-cactus-identification	Image	Classification
aptos2019-blindness-detection	Image	Ranking
denoising-dirty-documents	Image	Generation
detecting-insults-in-social-commentary	Text	Classification
dog-breed-identification	Image	Probabilistic Classification
dogs-vs-cats-redux-kernels-edition	Image	Probabilistic Classification
histopathologic-cancer-detection	Image	Classification
jigsaw-toxic-comment-classification-challenge	Text	Classification
leaf-classification	Image	Probabilistic Classification
mlsp-2013-birds	Audio	Classification
new-york-city-taxi-fare-prediction	Tabular	Regression
nomad2018-predict-transparent-conductors	Tabular	Regression
plant-pathology-2020-fgvc7	Image	Classification
random-acts-of-pizza	Text	Classification
ranzcr-clip-catheter-line-classification	Image	Classification
siim-isic-melanoma-classification	Image	Classification
spooky-author-identification	Text	Probabilistic Classification
tabular-playground-series-dec-2021	Tabular	Classification
tabular-playground-series-may-2022	Tabular	Regression
text-normalization-challenge-english-language	Text	Prediction accuracy (Seq2Seq)
text-normalization-challenge-russian-language	Text	Prediction Accuracy (Seq2Seq)
the-icml-2013-whale-challenge-right-whale-redux	Audio	Classification

Example 2: ModuleNotFoundError (`histopathologic-cancer-detection`). Baseline runs hit `ModuleNotFoundError` for `sklearn`/`tensorflow` in late steps; memory runs avoid these entirely.

Table 9: Late-phase error steps (step ≥ 10), averaged over 3 OpenHands runs per task. Δ = base $-$ mem (positive = memory helps).

Task	Base	Mem	Δ
aerial-cactus-identification	3.3	1.7	+1.7
aptos2019-blindness-detection	3.3	0.0	+3.3
denoising-dirty-documents	3.3	1.7	+1.7
detecting-insults-in-social-commentary	0.3	0.7	−0.3
dog-breed-identification	0.7	1.7	−1.0
dogs-vs-cats-redux-kernels-edition	0.7	0.0	+0.7
histopathologic-cancer-detection	1.7	0.3	+1.3
jigsaw-toxic-comment-classif.	0.0	0.0	0.0
leaf-classification	2.0	0.3	+1.7
mlsp-2013-birds	7.3	31.0	−23.7
new-york-city-taxi-fare-prediction	1.7	2.0	−0.3
nomad2018-predict-transparent-cond.	0.3	0.7	−0.3
plant-pathology-2020-fgvc7	0.3	2.7	−2.3
random-acts-of-pizza	20.7	1.0	+19.7
ranzcr-clip-catheter-line-classif.	0.3	1.3	−1.0
siim-isic-melanoma-classification	30.3	0.3	+30.0
spooky-author-identification	0.3	0.3	0.0
tabular-playground-series-dec-2021	0.7	0.7	0.0
tabular-playground-series-may-2022	6.3	0.0	+6.3
text-normalization-english	4.0	1.3	+2.7
text-normalization-russian	0.7	0.3	+0.3
the-icml-2013-whale-challenge	0.7	12.3	−11.7