Mistake Notebook Learning: Batch-Clustered Failures for Training-Free Agent Adaptation

Anonymous ACL submission

Abstract

With the growing adoption of Large Language Model (LLM) agents in persistent, real-world roles, they naturally encounter continuous streams of tasks and inevitable failures. A key limitation, however, is their inability to systematically learn from these mistakes, forcing them to repeat identical errors in similar contexts. Unlike prior training-free methods that primarily store raw instance-level experience or focus on retrieving successful trajectories, we propose Mistake Notebook Learning (MNL), a novel memory framework that enables agents to self-curate generalizable guidance from batch-clustered failures. This mechanism allows agents to distill shared error patterns into structured "mistake notes," updating an external memory only when batch performance improves to ensure stability. To further amplify adaptability, we integrate MNL with test-time scaling, leveraging aggregated failure patterns to actively steer the search process away from known pitfalls. Experiments on mathematical reasoning, Text-to-SQL, and interactive agent benchmarks show that MNL achieves competitive performance compared to existing memory mechanisms and in-context methods in both effectiveness and efficiency. These findings position structured mistake abstraction as a critical lever for robust agent evolution, enabling continuous improvement without the cost of parameter updates.

1 Introduction

Parameter-tuning is a standard approach for LLM adaptation but suffers from high computational costs, fragility to distribution shifts, and test-time rigidity in dynamic environments (Zeng et al., 2023; Chen et al., 2023; Zhai et al., 2025; Wang et al., 2024), hindering the rapid iteration essential for continual learning.

Training-free context methods offer an alternative, typically falling into two paradigms. Prompt-based optimization refines a single system prompt (Yang et al., 2024; Zhou et al., 2023; Pryzant et al., 2023) but often suffers from context length constraints and signal dilution. Memory-based approaches store instance-level experience (Shinn et al., 2023; Zhao et al., 2024; Zhang et al., 2024) to correct errors locally. However, they frequently lack subject-level abstraction, resulting in brittle behavior with limited generalization.

We introduce Mistake Notebook Learning (MNL), a training-free memory framework where the Tuner Model clusters failures by subject within batches via prompted subject clustering, distills shared error patterns into structured guidance, and commits updates only when batch performance improves. MNL positions adaptation as memory construction and context curation rather than weight updates, integrating with test-time scaling (TTS) to steer search away from systematically erroneous paths.

Across diverse domains including mathematics, Text-to-SQL, and agentic tasks, MNL demonstrates significant improvements with concise prompts and compact memory structures. Our experiments indicate that converting mistakes into generalized guidance serves as an effective lever for robust, low-overhead adaptation, achieving competitive performance compared to parameter-tuning baselines while maintaining efficiency.

Our contributions are threefold: (1) A general framework that enables evolution via batch-clustered mistake abstraction and structured guidance memory. (2) A conservative accept-if-improves rule that stabilizes memory evolution and prevents regressions. (3) Comprehensive validation across diverse domains—including mathematical reasoning, Text-to-SQL, and agentic workflows—demonstrating MNL's effectiveness and its compatibility with test-time scaling strategies.

2 Related Work

Agent Evolution and Memory Systems Strategies for agent evolution are generally categorized into parameter-tuning and training-free paradigms. Training-based methods, such as *AgentEvolver* (Zhai et al., 2025), *FireAct* (Chen et al., 2023), and *AgentTuning* (Zeng et al., 2023), typically rely on computationally intensive pipelines involving Supervised Fine-Tuning (SFT), Reinforcement Learning (RL), or evolutionary optimization (Qiu et al., 2025) to internalize capabilities into model weights. In contrast, Training-Free approaches leverage memory mechanisms to enable self-evolution without gradient updates. Memory modules have become a cornerstone in these systems, enabling agents to leverage historical context for enhanced decision-making (Wang et al., 2024). Contemporary memory systems adopt diverse storage formats, ranging from unstructured textual logs (Park et al., 2023) and latent vector embeddings to structured knowledge graphs. Recent advancements have further integrated Reinforcement Learning (RL) to optimize memory management policies (Yan et al., 2025; Wang et al., 2025a). For example, *Agentic Context Engineering (ACE)* (Zhang et al., 2025) treats context as an evolving "playbook," employing modular generation and reflection. *Memento* (Zhou et al., 2025) reframes continual learning as memory-based online reinforcement learning, employing a Case-Based Reasoning (CBR) mechanism to update memory without altering model parameters. Similarly, *Training-Free GRPO* (Cai et al., 2025) leverages group-relative semantic advantages to distill experiential knowledge into prompt-based token priors.

Learning from Mistakes Learning from mistakes is a critical capability for intelligent systems. Early works like *Reflexion* (Shinn et al., 2023) and *Self-Refine* (Madaan et al., 2023) utilize iterative verbal feedback to correct errors within a single session. However, these corrections are often transient and not retained for future tasks. To address this, recent research focuses on persistent learning. *LEAP* (An et al., 2024) and *CoTErrorSet* (Tong et al., 2024) explicitly fine-tune models on error-correction pairs to internalize mistake-avoidance capabilities. In the context of in-context learning, *ExpeL* (Zhao et al., 2024) and *In-Context Principle Learning* (Zhang et al., 2024) extract principles or rules from failures to guide future inference. While these methods demonstrate the value of negative feedback, they often treat mistakes as isolated instances or rely on static rule extraction.

Mistake Notebook Learning (MNL) Distinct from prior works that focus on retrieving successful trajectories or procedural workflows, MNL establishes a framework centered on *systematic mistake analysis*. While methods like ACE and Memento often operate at the instance level, MNL introduces a *batch-clustered* mechanism that aggregates errors to distill high-level, generalized insights, thereby reducing the variance associated with instance-specific corrections. Furthermore, we explore the integration of memory with *Test-Time Scaling (TTS)*. Unlike ReasoningBank (Ouyang et al., 2025), which enhances capabilities by retrieving successful reasoning traces, MNL synergizes its "Mistake Notebook" with TTS to actively mitigate potential errors. MNL demonstrates superior efficiency and adaptability in complex agentic workflows compared to vanilla scaling approaches.

3 Methodology

3.1 Method Overview

We propose **Mistake Notebook Learning (MNL)**, a memory-based, training-free, self-evolving framework designed to enhance the problem-solving proficiency of LLM-based agents. As illustrated in Figure 1, MNL operates with two distinct roles to enable evolution: the **Tuning Model** (π_θ), which generates responses and whose performance we aim to improve; and the **Tuner Model** (π_{tuner}), which analyzes failures and updates the memory. At its core, MNL maintains and continuously refines an external dynamic memory $\mathcal{M}$. Unlike prior approaches that accumulate instance-level experiences (Zhou et al., 2025; Zheng et al., 2024; Wang et al., 2025b), MNL leverages a **batch-clustered** mechanism: failed trajectories are clustered under shared semantic subjects by the Tuner Model via prompts, and generalized error patterns and corrective strategies are distilled, forming stable and transferable memory (Zhang et al., 2024). To ensure stability, updates are accepted only when they improve batch performance; otherwise, the previous memory state is retained. The framework follows a closed-loop process, iteratively performing baseline generation, memory update, and post-update evaluation to enable agents to self-evolve across different task domains and learning paradigms. The implementation details are presented in Appendix

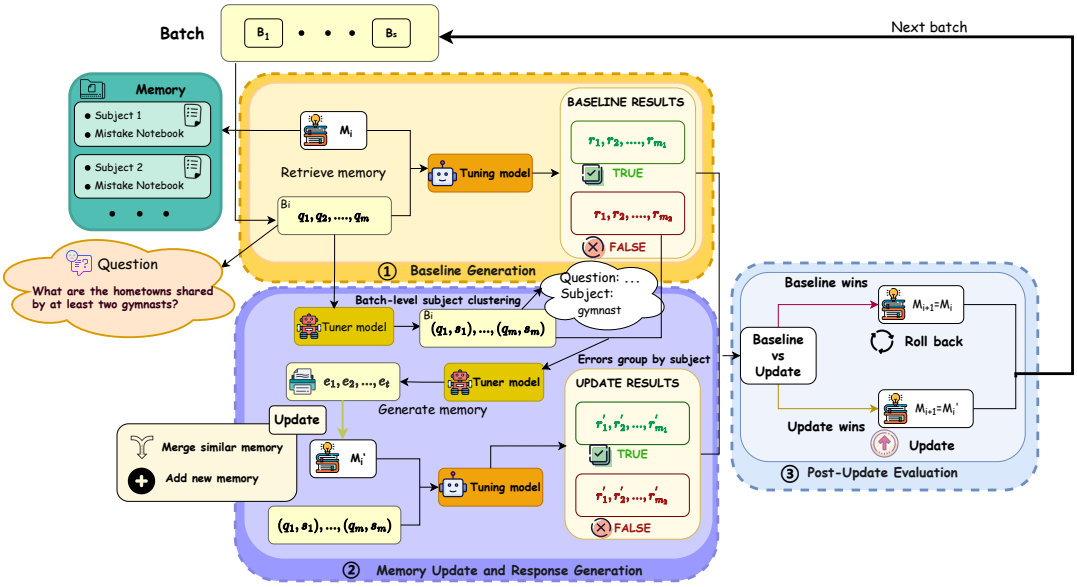
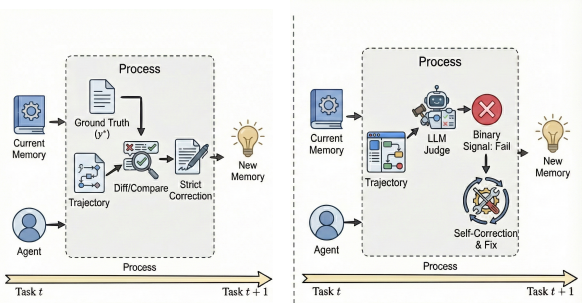


Figure 1: Overview of Mistake Notebook Learning (MNL). By utilizing a **Tuning Model** (the agent being improved) and a **Tuner Model** (the supervisor analyzing errors), the whole process consists of three steps: 1) Baseline Generation — The Tuning Model produces initial responses with the current prompt and memory to establish a performance baseline. 2) Memory Update and Response Generation — The Tuner Model performs batch-level subject clustering via prompts, analyzes baseline errors, creates structured guidance items, and selectively updates the memory. The Tuning Model then generates updated responses. 3) Post-Update Evaluation — Compare performance before and after the update to assess the effectiveness of the revised memory and decide whether to accept the update.



(a) Supervised Evolution (b) Self-Evolution

Figure 2: Comparison of the two operating regimes in MNL. (a) **Supervised Evolution** relies on explicit ground truth (y^*) for direct trajectory correction. (b) **Self-Evolution** leverages a proxy verifier (LLM Judge) to generate binary utility signals, enabling the agent to evolve solely from interaction experience without accessing ground truth labels.

A.2. The specific prompts utilized in this process are detailed in Appendix A.3.

As illustrated in Figure 2, we distinguish two *learning regimes*. MNL operates in a *Supervised Evolution* regime (Figure 2a), where explicit ground-truth answers are used to determine output correctness and provide feedback for memory construction. For agent tasks, MNL operates in a *Self-Evolution* regime (Figure 2b), in which a proxy verifier implemented as an LLM-based judge (Ouyang et al., 2025; Gu et al., 2025; Sun et al., 2025) assesses trajectory outcomes and produces binary utility signals, which enables memory generation without access to ground-truth labels, with the specific LLM judge prompts provided in Appendix A.4. Furthermore, we combine MNL memory with Test-Time Scaling (TTS) in agent tasks, performing memory induction on the test set prior to the final evaluation.

3.2 Problem Formulation

We formulate MNL as a context optimization problem aiming at constructing a semantic memory $\mathcal{M}$ that maximizes the expected reward of a frozen policy π_θ. Rather than updating model parameters, MNL improves performance by refining the memory $\mathcal{M}$ to ensure that the retrieved context $\text{Ret}(x, \mathcal{M})$ provides effective guidance for each input x.

Formally, for a task distribution $\mathcal{D} = \{(x, y)\}$,

we seek an optimal memory

$$\mathcal{M}^* = \arg\max_{\mathcal{M}} \mathbb{E}_{(x,y)\sim\mathcal{D}}\big[R(\pi_\theta(z), y)\big], \quad (1)$$

$$\text{where } z = x \oplus \text{Ret}(x, \mathcal{M}). \quad (2)$$

where $\oplus$ denotes prompt concatenation and $R(\cdot)$ is a reward function. In supervised settings, R is derived from ground-truth labels; in self-correction settings, it is estimated by a proxy verifier.

The optimization of $\mathcal{M}$ is delegated to a dedicated **Tuner Model** (π_{tuner}). Distinct from the inference role of **Tuning Model** π_θ, the Tuner Model acts as a reflective supervisor. It aggregates failed trajectories from π_θ, performs prompted subject clustering to identify systematic error patterns across batches, and synthesizes structured corrections to update the memory. This decoupling of execution (π_θ) and evolution (π_{tuner}) allows MNL to support various deployment configurations, such as self-correction (Madaan et al., 2023) or expert-guided distillation (Kim et al., 2025). Detailed prompts governing the Tuner Model's operations are provided in Appendix A.3.

Depending on the available resources, these roles can be instantiated in two *tuning configurations*: (1) **Self-Tuning**, where a single base model functions as both the Tuning Model and the Tuner Model to autonomously refine its own memory; and (2) **Cross-Model Tuning**, where a stronger model serves as the Tuner Model to distill high-quality guidance for a weaker Tuning Model. Table 7 later compares these two configurations on Qwen3-8B.

3.3 The MNL Evolution Protocol

As illustrated in Figure 1, the MNL framework operates through a closed-loop iterative process consisting of three sequential steps: Baseline Generation, Memory Update, and Post-Update Evaluation. This cycle ensures the continuous refinement of the memory $\mathcal{M}$ based on empirical performance feedback.

Step 1: Baseline Generation The process commences with the Tuning Model π_θ generating initial responses for a batch of queries. For each query x, the system retrieves relevant memory entries $\text{Ret}(x, \mathcal{M})$ to serve as advisory context. The Tuning Model is instructed to critically evaluate this context rather than blindly following it, thereby mitigating the risk of hallucination (see Appendix A.3.1). These initial responses establish a performance baseline for the current iteration.

Step 2: Memory Update and Response Generation The Tuner Model π_{tuner} analyzes the failed trajectories identified in the baseline generation. To solve the context optimization problem in Eq. (1)-(2), we propose a batch-clustered approach that extracts generalized error patterns from failed trajectories. At iteration t, we sample a batch $\mathcal{B} = \{(x_i, y_i)\}_{i=1}^{B} \sim \mathcal{D}$ and generate baseline outputs $\hat{y}_i = \pi_\theta\big(x_i \oplus \text{Ret}(x_i, \mathcal{M}_t)\big)$. Define the failure index set $\mathcal{F} = \{i \mid R(\hat{y}_i, y_i) = 0\}$ (or thresholded for real-valued rewards). A subject mapper $\sigma : \mathcal{X} \to \mathcal{S}$ is implemented by the Tuner Model, which performs semantic categorization of failed queries into precise subjects (e.g., combining domain, problem type, and solution method) via prompted clustering. This mapping, as detailed in Appendix A.3.2, induces subject clusters $S_s = \{i \in \mathcal{F} \mid \sigma(x_i) = s\}$ over the failure set. The tuner then extracts cluster-level guidance

$$g_s = \mathcal{E}\big(\{(x_i, y_i, \hat{y}_i)\}_{i\in S_s}; \mathcal{M}_t\big), \quad s \in \mathcal{S}_{\mathcal{F}}, \quad (3)$$

where $\mathcal{E}$ is the extraction operator that distills structured guidance from multiple failed trajectories within the same subject. and updates memory via

$$\mathcal{M}_{t+1} = \text{Update}\big(\mathcal{M}_t, \{(s, g_s)\}_{s\in\mathcal{S}_{\mathcal{F}}}\big). \quad (4)$$

This batch-level abstraction is coupled with the accept-if-improves criterion in Eq. (5) to ensure stable memory evolution. New memory nodes are integrated either by merging with existing similar entries or by appending them as new nodes (see Appendix A.3.3 and A.3.4). Following this update, the Tuning Model generates refined responses conditioning on the updated memory.

Step 3: Post-Update Evaluation To ensure the reliability of memory evolution, the system compares the performance of the updated responses against the baseline. Let $\Delta_{\mathcal{B}}$ denote the net improvement in batch accuracy:

$$\Delta_{\mathcal{B}} = \sum_{i=1}^{B} \Big(\mathbb{I}[R(\hat{y}_i') > R(\hat{y}_i)] - \mathbb{I}[R(\hat{y}_i') < R(\hat{y}_i)]\Big), \quad (5)$$

where $\hat{y}_i$ and $\hat{y}_i'$ correspond to the outputs before and after the update, respectively. The memory update is accepted if and only if $\Delta_{\mathcal{B}} > 0$; otherwise, the previous memory state is retained. This ensures that only beneficial updates are kept, preserving the integrity of the "Mistake Notebook".

4 Experiments

In this section, we empirically validate the effectiveness of Mistake Notebook Learning (MNL) across three modalities: mathematical reasoning, Text-to-SQL, and interactive agents. We evaluate both task performance and efficiency, reporting memory size and inference-time guidance-token length alongside accuracy or success metrics. We further study sensitivity to key design choices (e.g., batch-level abstraction and training epochs) and compare MNL with supervised fine-tuning and cross-model tuning.

4.1 Experimental Setup

We evaluate MNL on three reasoning modalities: Mathematical (AIME 2024/2025 (Mathematical Association of America, 2025), GSM8K (Cobbe et al., 2021)), Text-to-SQL (KaggleDBQA (Lee et al., 2021), Spider (Yu et al., 2019)), and Interactive Agent (Mind2Web (Deng et al., 2023), AppWorld (Trivedi et al., 2024)). Specifically, AIME utilizes DAPO-100 as the training set, comprising 100 problems randomly sampled from the DAPO-Math-17K dataset (Yu et al., 2025). Following Cai et al. (2025), Spider and GSM8K adopt their respective standard training and test splits, consistent with their official configurations: Spider uses 7,000 training examples and 1,034 development samples for evaluation, while GSM8K uses 7,473 training and 1,319 test samples. On KaggleDBQA, we use the 87 provided examples for training and evaluate on the 185 test samples. We employ Qwen3-8B (Yang et al., 2025), DeepSeek-V3.2-Exp (DeepSeek-AI, 2025), and Qwen3-Max (Team, 2025) as base models. Evaluation metrics include Pass@32 for AIME, execution accuracy (EA) for Text-to-SQL, as well as Task Success (TS) and Step Accuracy (SA) for agent benchmarks. Vanilla baselines follow standard prompting strategies per benchmark.[1] Unless otherwise specified, we adopt the *Self-Tuning* configuration (the tuner shares the same base model as the tuning model); Table 7 later compares this with *Cross-Model Tuning*. Detailed settings, datasets, and baselines are provided in Appendix A.5.1.

[1] Math and Text-to-SQL use Chain-of-Thought prompting (Wei et al., 2023); Mind2Web uses few-shot prompting aligned with prior work; AppWorld uses ReAct-style prompting (Yao et al., 2023).

4.2 Main Results: Effectiveness and Efficiency

We first present results on standard reasoning benchmarks (Math and Text-to-SQL) where MNL operates under *Supervised Evolution*, followed by interactive agent tasks under *Self-Evolution*.

Mathematical Reasoning Results Table 1 reports AIME 2024/2025 results. MNL improves or preserves accuracy across vanilla models while keeping memory compact. On Qwen3-8B, MNL achieves 33.0%/30.0% using 51 memory entries and a guidance-token length of 66.8, outperforming retrieval-heavy baselines (e.g., Memento, ACE) that rely on much longer contexts (3k–7k tokens). On DeepSeek-V3.2-Exp, MNL attains 90.0%/83.0% with 9 memory entries and a guidance-token length of 60; TFGO achieves slightly higher AIME accuracy but requires longer traces with a length of 696 tokens. On Qwen3-Max, MNL matches the vanilla model on both years (93.0%/96.0%), indicating no degradation on a highly capable base model. On GSM8K (Table 6), MNL improves accuracy by +2.1 points and narrows the gap to SFT to 0.4 points.

Text-to-SQL Results Table 1 reports execution accuracy on KaggleDBQA. Across base models, MNL improves over vanilla while keeping memory compact and guidance-token length moderate. The gains are most pronounced on DeepSeek-V3.2-Exp: MNL boosts EA from 24.0% to 64.0% using 54 memory entries and 514 guidance tokens, whereas ACE (Zhang et al., 2025) attains 54.0% but requires a much longer context of 9406 tokens. On Qwen3-8B, MNL reaches 28.0% with 752 guidance tokens, substantially shorter than ACE (6289 tokens) and far more accurate than vanilla (19.0%). On Qwen3-Max, MNL improves over vanilla (46.0% vs. 40.0%) with a compact memory and a short prompt compared to retrieval-heavy alternatives like Memento (Zhou et al., 2025). Table 6 further shows that MNL improves Spider accuracy over vanilla without parameter updates, narrowing the gap to SFT.

Interactive Agent Results Tables 2 and 3 show results on interactive agents under *Self-Evolution*. The main accuracy metrics include Task Success (TS) and Step Accuracy (SA). On Mind2Web, MNL improves Step Accuracy while reducing guidance-token length by orders of magnitude compared to retrieval/trajectory-heavy baselines (e.g., ACE (Zhang et al., 2025) and Memento (Zhou

Table 1: Main results on AIME 2024/2025 and KaggleDBQA. **Acc**: Pass@32 (AIME) / EA (KaggleDBQA). **Mem**: memory entries. **Len**: average guidance tokens (lower is better). Best in **bold**; "-" not applicable.

Method	AIME 2024 / 2025				KaggleDBQA		
	Acc-24 (%)	Acc-25 (%)	Mem	Len	EA (%)	Mem	Len
Qwen3-8B							
Vanilla	30%	23%	-	-	19%	-	-
TFGO	23%	23%	-	703	22%	-	**34**
Memento	20%	27%	100	3100	15%	87	530
ACE	27%	10%	100	7355	22%	98	6289
MNL	**33%**	**30%**	51	**67**	**28%**	50	752
DeepSeek-V3.2-Exp							
Vanilla	87%	80%	-	-	24%	-	-
TFGO	**93%**	**90%**	-	696	24%	-	**100**
ACE	80%	67%	163	21318	54%	96	9406
Memento	-	-	-	-	19%	87	1419
MNL	90%	83%	9	**60**	**64%**	54	514
Qwen3-Max							
Vanilla	**93%**	**96%**	-	-	40%	-	-
TFGO	90%	90%	-	1452	**47%**	-	**125**
Memento	-	-	-	-	**47%**	87	992
MNL	**93%**	**96%**	10	**0**	46%	54	375

Table 2: Results on interactive agent tasks on Mind2Web (%).

Method	TS (%)	SA (%)	Mem	Len
Qwen3-8B				
Vanilla model	1.35%	11.54%	-	-
Memento	0.00%	0.18%	1707	4749
ACE	0.00%	0.00%	363	24284
MNL	**2.02%**	**15.64%**	695	**556**
DeepSeek-V3.2-Exp				
Vanilla model	15.49%	66.32%	-	-
Memento	0.34%	12.60%	1707	4822
ACE	15.82%	57.80%	580	58602
MNL	**18.86%**	**67.55%**	12	**395**

Table 3: Results on interactive agent tasks on AppWorld (%).

Method	TS (%)	Mem	Len
Qwen3-8B			
Vanilla model	12.5%	-	-
Memento	12.5%	50	707
ACE	0.0%	61	3217
MNL	**14.3%**	12	**391**
DeepSeek-V3.2-Exp			
Vanilla model	**73.2%**	-	-
Memento	64.2%	56	602
ACE	44.6%	8	6902
MNL	**73.2%**	0	**0**

et al., 2025)). With DeepSeek-V3.2-Exp, MNL improves over vanilla on both Task Success and Step Accuracy (18.86/67.55 vs. 15.49/66.32) while using only 12 memory entries and 395 tokens; in contrast, ACE uses 58602 tokens. On AppWorld, MNL delivers low-overhead steering: on Qwen3-8B it improves Task Success (14.3 vs. 12.5) with 391 tokens, and on DeepSeek-V3.2-Exp it matches vanilla Task Success (73.2) while eliminating additional guidance tokens. Overall, MNL's batch-level mistake abstraction yields compact guidance that remains effective for multi-step interactions without inflating the prompt.

Table 4: Results on Mind2Web with think-enabled (TTS) models and Mimo-v2 variants (%).

Method	TS (%)	SA (%)	Mem	Len
Qwen3-8B-w/-think				
Vanilla model	1.01%	11.13%	-	-
MNL	**1.35%**	**12.60%**	695	**505**
Mimo-v2-w/o-think				
Vanilla model	**8.75%**	**41.12%**	-	-
MNL	8.08%	40.71%	410	**413**
Mimo-v2-w/-think				
Vanilla model	10.77%	47.63%	-	-
MNL	**11.09%**	**48.51%**	410	**423**

Integration with Test-Time Scaling (TTS) All main-table results use no-think base models. We additionally evaluate TTS-enabled variants (w/ think) in Tables 4 and 5. MNL remains compatible with TTS and provides consistent gains: As shown in Tables 4, on Mind2Web with a think-enabled Qwen3-8B model, Task Success improves from 1.01% to 1.35% and Step Accuracy from 11.13% to 12.60%; As shown in Tables 5, on AppWorld with DeepSeek-Reasoner, Task Success improves from 75.0% to 76.2%.

Table 5: Results on AppWorld with think-enabled (TTS) models and Mimo-v2 variants (%).

Method	TS (%)	Mem	Len
Qwen3-8B-w/-think			
Vanilla model	8.9%	-	-
MNL	**10.7%**	12	**391**
DeepSeek-Reasoner			
Vanilla model	75.0%	-	-
MNL	**76.2%**	4	**341**
Mimo-v2-w/o-think			
Vanilla model	69.6%	-	-
MNL	**71.4%**	1	**206**

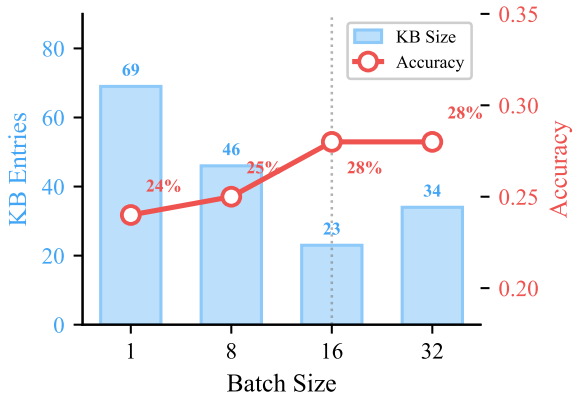


Figure 3: Effect of batch size on KaggleDBQA. Batch size 16 achieves optimal balance: 28% accuracy with only 23 KB entries vs. 24% accuracy with 69 entries at batch size 1.

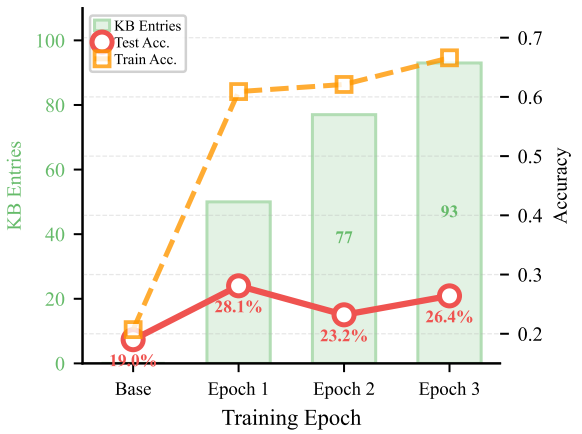


Figure 4: Effect of training epochs on KaggleDBQA. Single-epoch achieves optimal test accuracy (28.1%) with 50 KB entries. Multiple epochs cause cross-epoch overfitting: test accuracy drops to 23.2% at epoch 2 while training accuracy rises to 62.1%, demonstrating the memory overfits to training patterns.

Table 6: MNL vs. SFT on Qwen3-8B (Pass@1, %). Best results in **bold**.

Dataset	Vanilla	MNL	SFT
GSM8K	91.8%	93.9%	**94.3%**
Spider	68.9%	71.7%	**79.0%**

4.3 Analysis

Ablation Study: Batch-Level Abstraction Figure 3 confirms that batch-level abstraction reduces variance and improves generalization. Increasing batch size from 1 to 16 on KaggleDBQA improves accuracy by 17% while reducing memory size by a factor of 3. This validates our hypothesis that aggregating errors allows the model to distill more general principles rather than overfitting to isolated instances. Intuitively, clustering semantically related failures and averaging their signals reduces estimation noise, leading to more reliable memory updates (see Appendix A for theoretical analysis).

Ablation Study: Training Epochs Figure 4 shows that multi-epoch training leads to overfitting. Single-epoch training yields the highest test accuracy (28.1%), while subsequent epochs increase training accuracy but degrade test performance. This suggests that the "Mistake Notebook" is best constructed by seeing each error type once and generalizing, rather than repeatedly fitting to the training set. We thus adopt single-epoch training as a standard practice.

Comparison with Supervised Fine-Tuning Figure 5 (in Appendix) compares MNL with SFT on Qwen3-8B. Table 6 summarizes GSM8K and Spider results, showing that MNL narrows the gap to SFT on GSM8K and improves over the vanilla model on Spider without parameter updates.

Self-Tuning vs. Cross-Model Tuning We compare self-tuning (Qwen3-8B tuner) vs. cross-model tuning (DeepSeek-V3.2-Exp tuner) on Qwen3-8B. Table 7 shows that while cross-model tuning yields slightly higher performance on KaggleDBQA (31.0% vs. 28.0%), self-tuning remains competitive. This confirms MNL's practical applicability even when a stronger supervisor model is not available.

Training Cost Analysis Figure 6 highlights MNL's training-cost efficiency. On KaggleDBQA, MNL achieves 45.9% accuracy at \$0.19 training cost, half the training cost of Memento. On GSM8K/Spider, MNL approaches SFT accuracy at ∼40% lower training cost (see Appendix A.5.2). This makes MNL particularly attractive for budget-constrained deployment.

Table 7: Self-Tuning vs. Cross-Model Tuning on Qwen3-8B. Cross-Model Tuning (DeepSeek-V3.2-Exp as tuner) outperforms Self-Tuning, suggesting stronger tuner models can generate more effective guidance.

Dataset	Cross-Model	Self-Tuning
AIME 2025	30.0%	30.0%
KaggleDBQA	**31.0%**	28.0%

5 Conclusion

In this paper, we introduced Mistake Notebook Learning (MNL), a training-free framework that shifts LLM adaptation from parameter updates to structured memory and context curation. By leveraging batch-wise error abstraction and a selective accept-if-improves rule, MNL evolves a compact memory that steers frozen LLM behavior without gradient computation. We validated MNL under two regimes—Supervised Evolution with ground truth (Math and Text-to-SQL) and Self-Evolution with proxy judges (Mind2Web, AppWorld)—and observed consistent gains with short prompts and small memories. MNL is compatible with think-enabled models and enhances TTS performance, narrows the gap to SFT on GSM8K/Spider, and benefits from batch-level abstraction while avoiding multi-epoch overfitting. These results position memory- and context-centric adaptation as a practical, cost-efficient alternative to weight tuning for robust agent deployment.

Limitations

Retrieval and Subject Granularity MNL retrieves subject-level guidance via embedding similarity. Semantic asymmetry between concrete queries and abstract subjects can cause retrieval misses or mismatches, especially when subjects are overly broad or overly specific. Performance can therefore be sensitive to embedding quality, similarity thresholds, and the granularity of the subject taxonomy.

Feedback Quality and Verifier Reliability In supervised settings, memory construction depends on the availability and correctness of ground-truth signals. In self-evolution settings, proxy verifiers such as LLM judges may introduce bias or inconsistency, which can propagate into the memory and lead to suboptimal or unstable updates. Although the accept-if-improves rule mitigates regressions at the batch level, it cannot fully eliminate systematic verifier errors.

Scalability, Maintenance, and Safety As tasks and interactions grow, the memory can expand and increase retrieval and prompt-construction overhead. Additional mechanisms for memory consolidation and lifecycle management may be needed for long-running deployments. Finally, storing and reusing failure traces may raise privacy or safety concerns if trajectories contain sensitive information.

References

Shengnan An, Zexiong Ma, Zeqi Lin, Nanning Zheng, Jian-Guang Lou, and Weizhu Chen. 2024. Learning from mistakes makes llm better reasoner. *Preprint*, arXiv:2310.20689.

Yuzheng Cai, Siqi Cai, Yuchen Shi, Zihan Xu, Lichao Chen, Yulei Qin, Xiaoyu Tan, Gang Li, Zongyi Li, Haojia Lin, Yong Mao, Ke Li, and Xing Sun. 2025. Training-free group relative policy optimization. *Preprint*, arXiv:2510.08191.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. 2023. Fireact: Toward language agent fine-tuning. *Preprint*, arXiv:2310.05915.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

DeepSeek-AI. 2025. Deepseek-v3.2-exp: Boosting long-context efficiency with deepseek sparse attention.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: Towards a generalist agent for the web. *Preprint*, arXiv:2306.06070.

Jiawei Gu, Xinyi Jiang, Zichu Shi, Haoyu Tan, Xing Zhai, Can Xu, Wei Li, Yan Shen, Shuo Ma, Heng Liu, and 1 others. 2025. A survey on llm-as-a-judge. *Preprint*, arXiv:2411.15594.

Yujin Kim, Euiin Yi, Minu Kim, Se-Young Yun, and Taehyeon Kim. 2025. Guiding reasoning in small language models with llm assistance. *Preprint*, arXiv:2504.09923.

Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. 2021. KaggleDBQA: Realistic evaluation of text-to-SQL parsers. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2261–2273, Online. Association for Computational Linguistics.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. *Preprint*, arXiv:2303.17651.

Mathematical Association of America. 2025. American Invitational Mathematics Examination (AIME). https://artofproblemsolving.com/wiki/index.php/AIME_Problems_and_Solutions. Accessed: 2025-12-31.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. 2025. Reasoningbank: Scaling agent self-evolving with reasoning memory. *Preprint*, arXiv:2509.25140.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. *Preprint*, arXiv:2304.03442.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic prompt optimization with "gradient descent" and beam search. *Preprint*, arXiv:2305.03495.

Xin Qiu, Yulu Gan, Conor F. Hayes, Qiyao Liang, Elliot Meyerson, Babak Hodjat, and Risto Miikkulainen. 2025. Evolution strategies at scale: Llm fine-tuning beyond reinforcement learning. *Preprint*, arXiv:2509.24372.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Preprint*, arXiv:2303.11366.

Zeyi Sun, Ziyu Liu, Yuhang Zang, Yuhang Cao, Xiaoyi Dong, Tong Wu, Dahua Lin, and Jiaqi Wang. 2025. Seagent: Self-evolving computer use agent with autonomous learning from experience. *Preprint*, arXiv:2508.04700.

Qwen Team. 2025. Qwen3-max: Just scale it.

Yongqi Tong, Dawei Li, Sizhe Wang, Yujia Wang, Fei Teng, and Jingbo Shang. 2024. Can LLMs learn from previous mistakes? investigating LLMs' errors to boost for reasoning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3065–3080, Bangkok, Thailand. Association for Computational Linguistics.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. *Preprint*, arXiv:2407.18901.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6).

Yu Wang, Ryuichi Takanobu, Zhiqi Liang, Yuzhen Mao, Yuanzhe Hu, Julian McAuley, and Xiaojian Wu. 2025a. Mem-α: Learning memory construction via reinforcement learning. *ArXiv*, abs/2509.25911.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2025b. Agent workflow memory. In *Forty-second International Conference on Machine Learning*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Kristian Kersting, Jeff Z. Pan, Hinrich Schütze, Volker Tresp, and Yunpu Ma. 2025. Memory-r1: Enhancing large language model agents to manage and utilize memories via reinforcement learning. *Preprint*, arXiv:2508.19828.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2024. Large language models as optimizers. *Preprint*, arXiv:2309.03409.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, and 16 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *Preprint*, arXiv:2503.14476.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2019. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *Preprint*, arXiv:1809.08887.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. 2023. Agenttuning: Enabling generalized agent abilities for llms. *Preprint*, arXiv:2310.12823.

Yunpeng Zhai, Shuchang Tao, Cheng Chen, Anni Zou, Ziqian Chen, Qingxu Fu, Shinji Mai, Li Yu, Jiaji Deng, Zouying Cao, Zhaoyang Liu, Bolin Ding, and Jingren Zhou. 2025. Agentevolver: Towards efficient self-evolving agent system. *Preprint*, arXiv:2511.10395.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. 2025. Agentic context engineering: Evolving contexts for self-improving language models. *Preprint*, arXiv:2510.04618.

Tianjun Zhang, Aman Madaan, Luyu Gao, Steven Zheng, Swaroop Mishra, Yiming Yang, Niket Tandon, and Uri Alon. 2024. In-context principle learning from mistakes. *Preprint*, arXiv:2402.05403.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. Expel: Llm agents are experiential learners. *Preprint*, arXiv:2308.10144.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. 2024. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *International Conference on Representation Learning*, volume 2024, pages 19036–19066.

Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, and Jun Wang. 2025. Memento: Fine-tuning llm agents without fine-tuning llms. *Preprint*, arXiv:2508.16153.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large language models are human-level prompt engineers. *Preprint*, arXiv:2211.01910.

A Appendix

A.1 Why Batch-Level Abstraction Improves Decision Stability

We provide a brief proof sketch explaining why batch-level (cluster-level) abstraction can reduce the probability of spurious updates under the *accept-if-improves* criterion.

Setup. Consider a fixed subject s with cluster S_s. Let Δ_i denote the per-instance reward change induced by updating memory from $\mathcal{M}$ to $\mathcal{M}'$ with subject-level guidance:

$$\begin{aligned}\Delta_i =& R\big(\pi_\theta(x_i \oplus \text{Ret}(x_i, \mathcal{M}')), y_i\big) \\ &- R\big(\pi_\theta(x_i \oplus \text{Ret}(x_i, \mathcal{M})), y_i\big).\end{aligned} \tag{6}$$

For theoretical intuition, we assume an additive model for $i \in S_s$:

$$\Delta_i = \mu_s + \varepsilon_i, \quad \mathbb{E}[\varepsilon_i] = 0, \quad \text{Var}(\varepsilon_i) = \sigma_s^2 < \infty. \tag{7}$$

Here, μ_s captures the shared directional effect of the memory update on instances within the same semantic cluster, while ε_i models instance-specific noise. We assume independent noise among cluster members, which is reasonable when instances are grouped by semantic similarity rather than arbitrarily.

Cluster-Average Estimator. Define the cluster-average reward change:

$$\hat{\mu}_s = \frac{1}{|S_s|} \sum_{i \in S_s} \Delta_i. \tag{8}$$

Standard results imply $\hat{\mu}_s$ is unbiased:

$$\mathbb{E}[\hat{\mu}_s] = \mu_s,$$

with variance

$$\text{Var}(\hat{\mu}_s) = \frac{\sigma_s^2}{|S_s|}.$$

Implications for Accept-if-Improves. The *accept-if-improves* decision depends on the sign of the observed reward change. Let us consider the probability of an incorrect update decision given a true positive improvement $\mu_s > 0$:

$$\text{One-by-one: } \mathbb{P}(\Delta_i \le 0 \mid \mu_s > 0) = \mathbb{P}(\varepsilon_i \le -\mu_s), \tag{9}$$

$$\begin{aligned}\text{Cluster-avg: } &\mathbb{P}(\hat{\mu}_s \le 0 \mid \mu_s > 0) \\ &\le 2\exp\left(-c|S_s|\frac{\mu_s^2}{\sigma_s^2}\right),\end{aligned} \tag{10}$$

where the second line follows from standard concentration inequalities for sub-Gaussian noise ($c > 0$ is a constant).

Thus, using the cluster-average $\hat{\mu}_s$ exponentially reduces the probability of spurious sign flips compared to one-by-one updates. In other words, batch-level abstraction directly improves the reliability of the *accept-if-improves* decision rule.

A.2 Implementation Details

In this section, we provide a comprehensive overview of the MNL framework's implementation. We first define the structured Memory Schema

($\mathcal{M}$) in Appendix A.2.1, designed to store actionable and generalizable insights efficiently. We then detail the technical execution of the MNL Evolution Protocol in Appendix A.2, describing how the three-stage cycle of baseline generation, memory update, and post-update evaluation is realized in practice. Finally, we present the formal Algorithm that orchestrates this iterative MNL Evolution process in Appendix A.2.3.

A.2.1 Memory Schema and Storage

To ensure scalability and efficient retrieval, we maintain the memory $\mathcal{M}$ in a JSONL format, where each entry is defined as a structured tuple $e = \langle s, g, \phi(s) \rangle$. The **Subject** ($s$) serves as a high-level semantic cluster identifier (e.g., "SQL: Join conditions on null values") to facilitate broad topic matching. The **Memory** (g) comprises five mandatory components to ensure actionability and safety: (1) *Corrected Examples* that provide explicit mistake-answer pairs to ground the abstraction; (2) a *Correct Approach* detailing the step-by-step reasoning methodology; (3) a *Mistake Summary* identifying the root cause of the error; (4) a *Generalizable Strategy* summarizing reusable problem-solving patterns; and (5) *Anti-Patterns*, which are critical warnings specifying misapplication scenarios to prevent over-generalization. Finally, the **Embedding** $\phi(s)$ represents the semantic vector of the subject, pre-computed to enable efficient cross-modal retrieval against incoming query embeddings.

A.2.2 MNL Evolution Protocol Implementation Details

Baseline Generation. The process commences with a retrieval-augmented generation step. For a batch of incoming queries, we compute query embeddings and perform a similarity search against the subject embeddings $\phi(s)$ in $\mathcal{M}$, retrieving the top-k entries where the cosine similarity exceeds a specific threshold. These retrieved memory items are concatenated into the system context. To mitigate the risk of the Tuning Model π_θ blindly following potentially irrelevant historical advice, we append a specific applicability assessment instruction (see Appendix A.3.1). This compels the model to critically evaluate the relevance of the retrieved guidance before generating the initial baseline responses.

Memory Update and Response Generation. Following baseline generation, we employ a filtering mechanism to identify high-value learning opportunities. For domains with deterministic answers (e.g., Text-to-SQL, Math), correctness is determined by ground truth comparison; for open-ended agentic tasks, we utilize an LLM-as-a-judge to analyze the trajectory and produce binary success/failure signals (see Appendix A.4). The update process operates at the subject level rather than the instance level. We first employ a *Subject Clustering* step (prompt in Appendix A.3.2) to group failed queries into semantic clusters. The Tuner Model π_{tuner} then analyzes the collective failure trajectories within each cluster to distill the structured five-part memory described in Appendix A.2.1. To consolidate these insights into $\mathcal{M}$, we calculate the semantic similarity between the new subject and existing memory nodes. If the similarity exceeds a merge threshold, the new insights are fused into the existing node to refine the strategy; otherwise, a new node is appended.

Post-Update Evaluation. To guarantee the reliability of the evolving memory, we implement a closed-loop verification mechanism. The batch of queries is re-processed using the updated memory $\mathcal{M}'$, and we calculate the net performance improvement $\Delta_{\mathcal{B}}$ (see Eq. (5)). The memory update is committed only if $\Delta_{\mathcal{B}} > 0$; otherwise, the system rolls back to the previous state, ensuring that the memory $\mathcal{M}$ accumulates only beneficial and experimentally validated guidance.

A.2.3 The MNL Evolution Algorithm

Algorithm 1 presents the complete pseudocode for MNL, which consists of three main steps: (1) Baseline Generation, (2) Memory Update and Response Generation, (3) Post-Update Evaluation.

A.3 Prompts Used in MNL Implementation

A.3.1 Applicability Assessment Prompt

To prevent the model from blindly adopting retrieved memories that may be contextually mismatched, we prepend this instruction to the system prompt, enforcing a critical relevance check.

Applicability Assessment Prompt

The following mistake notes are not necessarily tied to the current question, but you may use them to deepen your analytical approach.

Algorithm 1 Mistake Notebook Learning (MNL)

Require: Task distribution $\mathcal{D}$, Tuning Model π_θ, Tuner Model π_{tuner}, Reward function R, Batch size B

Initialize global memory $\mathcal{M} \leftarrow \emptyset$
for each batch $\mathcal{B} = \{(x_i, y_i)\}_{i=1}^{B} \sim \mathcal{D}$ **do**
  // Step 1: Baseline Generation
  $\mathcal{Y}_{\text{base}} \leftarrow \emptyset$
  for $i = 1$ **to** B **do**
    $c_i \leftarrow \text{Ret}(x_i, \mathcal{M})$
    $z_i \leftarrow x_i \oplus c_i$
    $\hat{y}_i \leftarrow \pi_\theta(z_i)$
    $\mathcal{Y}_{\text{base}} \leftarrow \mathcal{Y}_{\text{base}} \cup \{\hat{y}_i\}$
  end for
  // Step 2: Memory Update and Response Generation
  $\mathcal{F} \leftarrow \{i \mid \hat{y}_i \text{ is identified as a failure}\}$
  if $\mathcal{F} = \emptyset$ **then**
    continue
  end if
  $\mathcal{G}_{\text{fail}} \leftarrow \text{ClusterFailuresBySubject}(\{(x_i, \hat{y}_i)\}_{i \in \mathcal{F}}, \pi_{\text{tuner}})$
  $\mathcal{M}' \leftarrow \mathcal{M}$ {Initialize candidate memory}
  for each subject group $S \in \mathcal{G}_{\text{fail}}$ **do**
    $\mathcal{P}_S \leftarrow \text{DistillPatternsAndStrategies}(S, \pi_{\text{tuner}})$
    $\mathcal{M}' \leftarrow \text{UpdateMemory}(\mathcal{M}', \mathcal{P}_S, \text{method} = \text{MergeOrAppend})$
  end for
  $\mathcal{Y}_{\text{new}} \leftarrow \emptyset$
  for $i = 1$ **to** B **do**
    $\hat{y}'_i \leftarrow \pi_\theta(x_i \oplus \text{Ret}(x_i, \mathcal{M}'))$
    $\mathcal{Y}_{\text{new}} \leftarrow \mathcal{Y}_{\text{new}} \cup \{\hat{y}'_i\}$
  end for
  // Step 3: Post-Update Evaluation
  $\Delta_{\mathcal{B}} \leftarrow \sum_{i=1}^{B} \Big(\mathbb{I}[R(\hat{y}'_i) > R(\hat{y}_i)] - \mathbb{I}[R(\hat{y}'_i) < R(\hat{y}_i)] \Big)$
  if $\Delta_{\mathcal{B}} > 0$ **then**
    $\mathcal{M} \leftarrow \mathcal{M}'$ {Accept evolution}
  else
    Discard $\mathcal{M}'$ {Retain previous state}
  end if
end for
return $\mathcal{M}$

IMPORTANT: Before applying any guidance below, carefully evaluate:

1. Does the current problem match the applicability conditions stated in the guidance?
2. Is the problem type and context similar to the examples in the guidance?
3. If the problem is fundamentally different (e.g., combinatorics vs modulo arithmetic, complex numbers vs number theory), do NOT force-fit the guidance.
4. Only use guidance that is clearly relevant to the current problem structure and requirements.

Before solving, review the attached guidance. State whether it is: "applicable", "partially applicable", or "irrelevant". Use only applicable parts when answering.

A.3.2 Subject Clustering Prompt

We cluster each question into a high-specificity subject for RAG retrieval:

Subject Clustering Prompt

You are an expert in categorizing questions into precise, high-relevance subjects for Retrieval-Augmented Generation (RAG). Your goal is to assign each question a subject label that:

- Maximizes retrieval relevance by precisely describing the problem type and solution method
- Groups only genuinely similar questions together (same domain AND same approach)
- Avoids over-broad categories that would match unrelated problems

Include: (1) Mathematical Domain, (2) Problem Type, (3) Solution Method.
Examples of GOOD subjects:

- "Combinatorics: Counting arrangements in grids with row and column sum constraints using stars and bars"
- "Complex Analysis: Evaluating products over roots of unity using polynomial evaluation"

Examples of BAD subjects (too broad):

- "modulo arithmetic" – could match any modulo problem
- "number theory" – could match any number theory problem

A.3.3 Structured Guidance Extraction Prompt

We derive structured batch-level clustered memory with a prompt template designed to capture our five-component memory representation.

Structured Guidance Extraction Prompt

You are an expert in analyzing model errors and maintaining a "mistake notebook" to improve future performance.
Subject: {subject}
Error Examples:
{error_context}
Task: Extract insights from the mistakes and rewrite them as a structured mistake note.
Your response must include :
1. Corrected Examples with mistake answers

- For each, include:
  - The original question and mistake answer
  - Correct answer and correct reasoning process

2. Correct Approach

- Provide the correct reasoning method or step-by-step approach that should be applied.

3. Mistake Summary

- Identify the root cause behind the errors (reasoning flaw, misunderstanding of concept, missing steps, incorrect logic, etc.).

4. Generalizable Strategy

- Summarize reusable problem-solving patterns and how to avoid future mistakes.

5. ANTI-PATTERNS
List specific things to AVOID:

- Common ways this guidance gets misapplied
- Situations where following this guidance would be WRONG
- Red flags that indicate the guidance doesn't fit

Output format should resemble a mistake notebook entry: concise, structured, knowledge-focused, and reusable for similar future questions.

A.3.4 RAG-Based Guidance Merging Prompt

We merge new memory with related existing entries to enable memory updating:

RAG-Based Guidance Merging Prompt

You are synthesizing guidance for subject: {subject}
Existing guidance from related subjects in the memory:
{existing_guidance}
New guidance to incorporate:
{new_guidance}
Task: Merge these into a single coherent guidance that:

- Combines insights from related subjects with new guidance
- Eliminates redundancy while preserving key information and examples of the mistakes
- **Preserves and emphasizes applicability conditions**—clearly state when each method applies
- Focuses on actionable advice
- Maintains consistent style
- **Includes warnings about when NOT to apply the guidance** to avoid misapplication

Merged guidance:

A.4 LLM Judge Prompts

To evaluate agent performance across different environments without relying solely on ground truth, we design specialized LLM-as-a-judge prompts. We tailor these prompts to the specific granularities of the Mind2Web and AppWorld benchmarks.

A.4.1 Mind2Web Evaluation Prompts

For the Mind2Web benchmark, we employ two distinct judging mechanisms. The Pairwise Comparison Judge (Appendix A.4.1) is utilized when the agent generates multiple candidate actions; it analyzes two options simultaneously to identify the optimal next step based on UI logic. Conversely, the Single Trajectory Judge (Appendix A.4.1) acts as a binary verifier, analyzing a specific action in isolation to determine its validity within the interaction flow.

Pairwise Comparison Judge Prompt

You are an expert in web navigation and user interface interaction.
Given this web navigation task: {question}
Compare these two proposed actions and determine which one is MORE CORRECT:
Action A: {answer1} Action B: {answer2}
Evaluation criteria (in order of importance):
Task relevance - Does this action directly help achieve the stated goal?
UI logic - Is this a logical next step given the current page state?
Element availability - Does the target element actually exist on the page?
Efficiency - Is this the most direct path to accomplish the task?
Think step by step, then respond with exactly ONE of these options:
"Action A is more correct"
"Action B is more correct"
"Both are equally correct or equally wrong"
Your response must start with one of these exact phrases.

Single Trajectory Judge Prompt

You are an expert in web navigation and user interface interaction evaluation. Your task is to determine if a candidate answer is correct for a given web navigation task.
You DO NOT have access to a ground truth answer, so you must judge strictly based on the provided web context (HTML), the user's task goal, and the interaction history.
Context and Task: {question}
Proposed Action to Evaluate: {candidate_answer}
Evaluation Steps:
Goal Analysis: What is the user trying to achieve?
State Analysis: Based on previous actions, where are we in the flow?
Element Verification: Does the element selected in the proposed action exist in the HTML? Is it the correct element to interact with?
Action Validity: Is the action (CLICK, TYPE, SELECT) appropriate for this element and goal?
Judgment Criteria:
CORRECT: The action is the logical, necessary, and correct next step to advance the task.
INCORRECT: The action is irrelevant, interacts with the wrong element, uses the wrong action type, or hinders the task.
Respond with exactly ONE of the following lines, followed by your reasoning:
"Judgment: CORRECT"
"Judgment: INCORRECT"

A.4.2 AppWorld Evaluation Prompt

Unlike the step-by-step UI interactions in Mind2Web, the AppWorld benchmark requires evaluating complete API interaction chains and Python code execution. Therefore, the AppWorld Judge (Appendix A.4.2) is designed to assess the full execution log, determining success based on the final output validity and the absence of fatal runtime errors.

AppWorld Execution Judge Prompt

You are an expert Judge for an AI Agent execution log. Your goal is to determine if the Agent successfully completed the user's task based on the provided execution log.
Input Data:
1. Task Question: "{question}"
2. Execution Log (JSON format):
«<EXECUTION_LOG_START»>
{trajectory}
«<EXECUTION_LOG_END»>

Judgment Criteria:
- **SUCCESS**: The agent found the requested information, performed the requested action, or provided the correct answer in the final turn. The code executed without fatal errors.
- **FAILURE**: The agent encountered a Python exception that stopped progress, got stuck in a loop, failed to call the correct API, authorized incorrectly, or the log ends abruptly without an answer.
Output Format:
You must output a single JSON object with the following structure:
"reasoning": "Brief analysis of why it succeeded or failed (max 50 words).", "is_success": true or false

A.5 EXPERIMENT DETAILS

A.5.1 Evaluation Protocol and Experimental Settings

Evaluation Protocol. We evaluate all settings under greedy decoding with **temperature** set to 0.0. We adopt task-dependent Pass@k: for **mathematical reasoning**, we report **Pass@32** on **AIME 2024/2025** to mitigate sampling variance on challenging problems, while all other tasks (including GSM8K, Text-to-SQL, and agent benchmarks) are evaluated with **Pass@1**.

Evaluation metrics are task-specific: for **Text-to-SQL**, we report *execution accuracy*, where a predicted SQL query is executed on the target database and matched against the gold execution result; for **mathematical reasoning**, we use *normalized exact match* after symbolic simplification; for **agent tasks**, we evaluate **Mind2Web** using *Task Success* and *Step Accuracy*, and **AppWorld** by whether the agent successfully completes the user-defined *task goal* in the real application environment. This "task-specific metrics" framing follows common practice in agent evaluation setups.

Detailed Dataset Statistics and Splits. **AIME 2024/2025** training uses 100 examples randomly sampled from DAPO-Math-17K, and the test sets contain 30 problems each for AIME 2024 and AIME 2025. **GSM8K** uses 7,473 training examples and 1,319 test examples. **KaggleDBQA** uses 87 training examples and 185 test examples. Spider uses 7,000 training examples and 1,034 test examples. For **Mind2Web**, we select 3 subjects; for each subject we randomly sample 100 tasks. Each task contains average 6 steps, resulting in 1,707 training instances and 1,707 test instances. For **AppWorld**, we randomly sample one instance from each of the 56 scenarios for both training and testing, yielding 56 training instances and 56 test instances.

Table Conventions. In all result tables, higher accuracy/success is better; best performance values are marked in **bold**. **Mem Cnt** denotes the number of memory entries constructed during training, and **Avg. Len (tok)** denotes the average number of guidance tokens used during inference. **Mem Cnt** is not applicable to TFGO due to prompt-based optimization. For cost reasons, we omit ACE results with Qwen3-Max where applicable; similarly, Memento and TFGO are omitted in some large-scale experiments due to their high token consumption and associated budgetary constraints.

Main Result Notes. On AIME 2024/2025, MNL improves or preserves accuracy across base models with small memory. On KaggleDBQA, MNL yields consistent accuracy gains over the vanilla model and avoids the large context overhead of retrieval-heavy baselines. On Mind2Web and AppWorld, MNL improves agent success under self-correction while keeping inference context short.

Following our ablation results, all experiments use **single-epoch** training to avoid cross-epoch overfitting. Unless otherwise specified, we adopt the **Self-Tuning** setting, where the tuner shares the same architecture as the model being tuned. We use model-specific maximum generation lengths: **Qwen3-8B** and **Qwen3-Max** use a 32K-token limit, while **DeepSeek-V3.2-Exp** supports up to 8K tokens and is therefore evaluated with an 8K limit. All vanilla models are evaluated under the **no-think** setting.

Implementation Details. For **Text-to-SQL**, **mathematical reasoning**, we use an **supervised evolution** setting, i.e., ground-truth answers are available during training/tuning to support explicit error attribution and feedback construction. In contrast, for **Mind2Web** and **AppWorld** we use an **self-evolution** setting: no ground-truth answers are provided during training, and an **LLM Judge** determines the agent's *task success* based on the final interaction outcome, which is then used to generate feedback signals for self-tuning. Finally, all meth-

ods (including TFGO, ACE, Memento, and MNL) are evaluated under the same protocol with ground true to ensure fair and reproducible comparison.

Reproducibility Settings. To ensure reproducibility, we fix the hyperparameters to Temperature=0, Presence Penalty=1.5, and Random Seed=42. For Qwen3-8B, we use max-tokens=32K; for DeepSeek and Qwen3-Max, we set max-tokens=8K. To better match different evaluation settings, we further set max-tokens=8K for Text-to-SQL and all ablation studies. For method-specific configurations, **MNL** uses epoch=1 and batch-size=16, with bge-m3 as the embedding model; during retrieval we set topk=1 and retrieval-threshold=0.6. **Memento** sets memory-max-pos-examples=4, memory-max-neg-examples=4, and memory-max-length=256, and also uses bge-m3 as the embedding model. For Memento, META-MODEL, EXEC-MODEL, and JUDGE-MODEL share the same backbone model. **TFGO** uses batchsize=64, rollout-concurrency=5, and rollout-max-tokens=4096. ACE uses epoch=1, max-num-rounds=3, and playbook-token-budget=80K; generator-model, reflector-model, and curator-model are instantiated with the same backbone model.The experimental settings for the other methods largely follow the default configurations provided in their open-source implementations.

A.5.2 Cost Calculation Details

For KaggleDBQA, we use Qwen3-Max as the self-tuning mode and compute learning cost based on its official API pricing. Specifically, the pricing for Qwen3-Max-3 is 0.0032 RMB per 1k input tokens and 0.0128 RMB per 1k output tokens. The total learning cost is obtained by aggregating the number of input and output tokens generated during training according to these rates. For GSM8K and Spider, we use Qwen3-8B as the base model and conduct training on a single H20 GPU (141GB). The GPU usage cost is computed at a rate of $3.99 per hour. GPU prices follow the publicly listed on-demand pricing at the time of experiments. MNL+Qwen3-8B completes training in 15 minutes on GSM8K, resulting in a total learning cost of $0.99, while SFT+Qwen3-8B requires 30 minutes of training under the same hardware configuration, incurring a cost of $1.98. On Spider, MNL+Qwen3-8B completes training in 30 minutes with a cost of $1.98, whereas SFT+Qwen3-8B requires 50 minutes of training time and incurs a cost of $3.32 under identical computational resources. All reported costs account for training only and exclude inference or evaluation overhead.

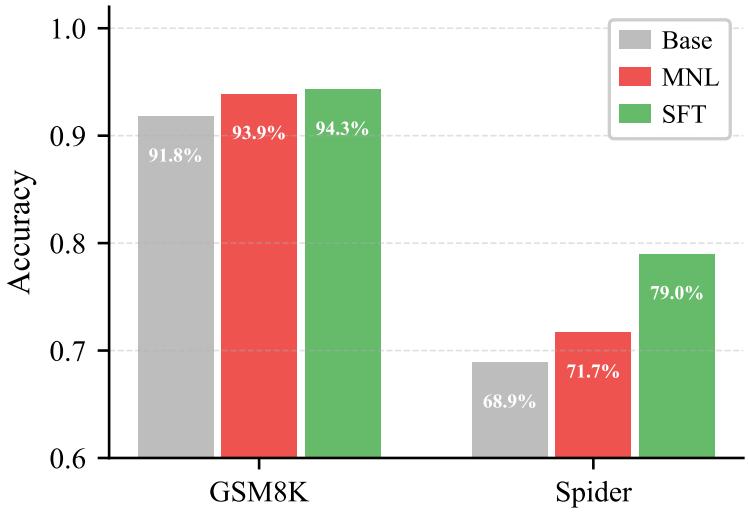


Figure 5: MNL vs. SFT on Qwen3-8B. On GSM8K, MNL (93.9%) nearly matches SFT (94.3%). On Spider, SFT (79.0%) leads, but MNL (71.7%) improves over Vanilla model (68.9%) without parameter updates.

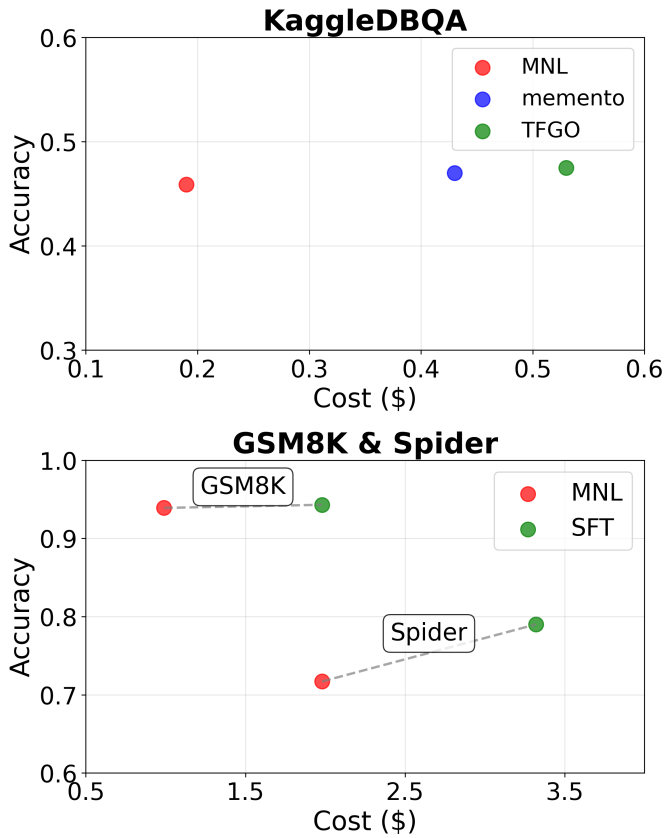


Figure 6: Cost-accuracy trade-off. Top: On KaggleDBQA, MNL achieves 45.9% accuracy at \$0.19, while Memento reaches 47.0% accuracy at \$0.43 (2.3× cost). Bottom: On GSM8K/Spider, MNL approaches SFT accuracy at 40% lower cost.