Experience-Guided Adaptation of Inference-Time Reasoning Strategies

Adam Stein[1]*, Matthew Trager[2], Benjamin Bowman[2], Michael Kleinman[2], Aditya Chattopadhyay[2], Wei Xia[2], Stefano Soatto[2]

[1]University of Pennsylvania, [2]AWS AI

Abstract

Enabling agentic AI systems to adapt their problem-solving approaches based on post-training interactions remains a fundamental challenge. While systems that update and maintain a memory at inference time have been proposed, existing designs only steer the system by modifying textual input to a language model or agent, which means that they cannot change sampling parameters, remove tools, modify system prompts, or switch between agentic and workflow paradigms. On the other hand, systems that adapt more flexibly require offline optimization and remain static once deployed. We present Experience-Guided Reasoner (EGuR), which generates tailored *strategies*—complete computational procedures involving LLM calls, tools, sampling parameters, and control logic—dynamically at inference time based on accumulated experience. We achieve this using an LLM-based *meta-strategy*—a strategy that outputs strategies—enabling adaptation of all strategy components (prompts, sampling parameters, tool configurations, and control logic). EGuR operates through two components: a **Guide** generates multiple candidate strategies conditioned on the current problem and structured memory of past experiences, while a **Consolidator** integrates execution feedback to improve future strategy generation. This produces complete, ready-to-run strategies optimized for each problem, which can be cached, retrieved, and executed as needed without wasting resources. Across five challenging benchmarks (AIME 2025, 3-SAT, and three Big Bench Extra Hard tasks), EGuR achieves up to 14% accuracy improvements over the strongest baselines while reducing computational costs by up to 111×, with both metrics improving as the system gains experience.

1 Introduction

Modern AI systems employ sophisticated *strategies*—complex computational procedures involving LLM calls, tools, and control logic, often implemented using frameworks like SGLang [1] or DSPy [2]—to tackle challenging reasoning tasks. For example, systems solving International Mathematical Olympiad (IMO) problems dynamically decompose problems into subgoals, verify solutions through parallel sampling, and iteratively self-correct [3, 4]. Similarly, code generation agents switch between different prompting approaches, adjust sampling temperatures, and selectively use tools based on problem complexity [2, 5].

While these strategies are often designed to be general-purpose and highly adaptive to inputs—particularly agents, which consist of loops of LLM and tool execution—they generally remain *static* at inference time, unable to learn from experience. For example, without persistent memory, an AI system for mathematical reasoning will apply the same expensive multi-step decomposition every time it encounters a problem it has solved in the past (potentially even failing to solve it again), and will repeatedly make identical mistakes on problems it has failed to solve before.

To address this problem, different methods for adapting AI systems have been proposed. Approaches such as Dynamic Cheatsheet [6] and Buffer of Thoughts [7] maintain memory across problems, but this memory is used only for *textual steering* of an LLM or agent. As such, they cannot change sampling parameters, add or remove tools, modify control logic, or cache successful computational procedures for direct

*Work done during an internship at AWS AI. Correspondence to: Matthew Trager <mttrager@amazon.com>.

reuse. On the other hand, offline methods like ADAS [8] can adapt strategies much more flexibly, but require expensive training phases and remain static once deployed, unable to continue learning from new experiences. The core challenge is to **adapt the computational structure of strategies from experience, to continually improve both accuracy and efficiency**.

To address this challenge, we present Experience-Guided Reasoner (EGUR), a system designed to generate strategies dynamically at inference time. Our approach uses an LLM-based *meta-strategy*—a strategy that outputs strategies—rather than steering existing strategies at runtime, it proposes new strategies. This enables generation of complete, tailored computational procedures for each problem that can be easily conditioned using a memory store and enables simple and effective caching of successful strategies for reuse. EGUR operates through two main components: a **Guide** that generates multiple candidate strategies conditioned on the current problem and structured memory of past experiences, and a **Consolidator** that processes execution outcomes to improve future strategy generation. By generating and comparing multiple strategies per problem, the system learns the relative effectiveness of different strategies, leading to continual improvements in both accuracy and computational efficiency.

Our contributions are:

- We introduce a system that generates strategies dynamically at inference time based on accumulated experience, enabling adaptation of all strategy components including prompts, sampling parameters, tools, and control logic (Section 3).
- We formalize strategies as compositions of stateful *processes* which map inputs to outputs while also updating a state. This provides a unified representation for diverse reasoning paradigms (workflows, agents, tool-use systems) that supports compositional cost tracking and execution tracing (Section 2.1).
- We demonstrate that generating multiple strategies per problem and comparing their effectiveness enables continual improvement in both accuracy and efficiency (Figure 5).
- Across five challenging benchmarks (AIME 2025, 3-SAT, and three Big Bench Extra Hard tasks), EGUR achieves up to 14% accuracy improvements (EGUR vs. Mem0 on 3-SAT) while reducing computational costs by up to 111x (EGUR vs. DC on Object Counting), with both metrics improving as the system gains experience (Figure 4).

2 Strategies For AI Systems

In this section, we formalize strategies as compositions of stateful processes—functions taking inputs and a state and producing an output and an updated state—providing a unified way to describe any strategy in terms of its components. This yields a natural taxonomy of common strategies and enables us to describe feedback mechanisms as state-modification methods.

2.1 Representing Strategies

We use the term *strategy* for a specification of the computational procedure that an AI system applies to inputs to produce outputs, capturing LLM inference calls, tool invocations, and control flow decisions. For example, the Chain-of-Thought (CoT) strategy involves a single LLM call with specific prompting, while CodeAct iteratively calls an LLM and executes generated code until a termination condition is met.

An adaptive strategy engages in a sequence of interactions, or episodes, and updates its internal state based on past outcomes. To enable adaptive strategies, we need a representation that explicitly captures both what can be adapted and how state evolves across episodes. Existing frameworks like DSPy [2] and SGLang [1] implement strategies as Python code without explicitly marking adaptable components. TextGrad [5] and Trace [10] mark components for optimization, but are designed for offline training and lack explicit state management for continual learning during inference. We introduce a compositional formalism that unifies adaptable components with state management, enabling meta-strategies to generate and refine strategies from accumulated experience.

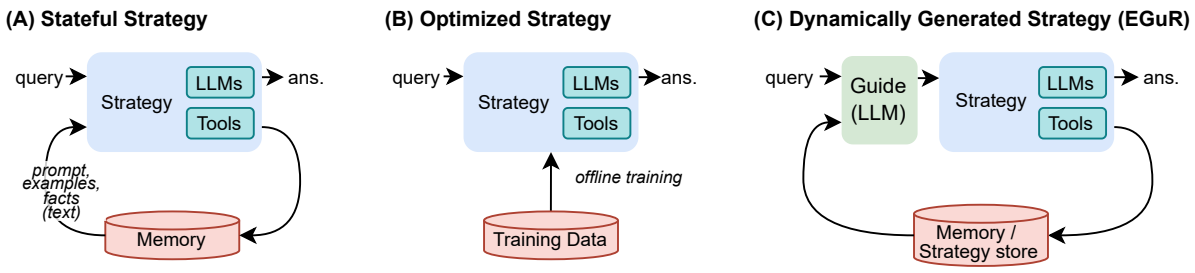

Figure 1: Comparison of existing experience-based adaptation methods (A and B) to our approach in (C) which dynamically produces a compiled strategy based on the current query and memory at inference time. Methods which augment existing strategies with state, such as Dynamic Cheatsheet[6] and Mem0[9], are depicted in (A) and methods such as ADAS [8] and OPTO [10] which optimize strategies offline are shown in (B). Our method shown in (C) uses a state (which can contain useful compiled strategies) during inference to guide the system in producing effective strategies for each query, unlike existing methods which cannot adapt their strategies per-query.

We model strategies as compositions of stateful *processes*. A process is a function that takes an input and a state (e.g., conversation history, sampling parameters), produces an output, and updates the state. This explicit state management clarifies what components can be adapted (prompts, temperature, available tools). Formally, we write a process type as $A \overset{\Sigma}{\Longrightarrow} B : (A \times \Sigma) \to (B \times \Sigma)$ where A and B are input and output domains and Σ is the state space. Strategies are built compositionally using the grammar shown in Figure 2. Sequential composition (S_1; S_2) chains processes, parallel composition ($S_1 \parallel S_2$) executes concurrently, conditionals enable branching, and recursion enables iteration.

For instance, CodeAct (shown in Figure 2b) composes an LLM call with a code execution tool recursively, iterating until the LLM output contains no code. The LLM call process maintains conversation history in its state, while the code execution process maintains the execution environment.

This compositional structure is key to our approach. By explicitly representing all components (prompts, sampling parameters, tools, control flow) and their state dependencies, we enable a meta-strategy to generate new strategies by composing different components. For instance, our meta-strategy can adapt CodeAct by changing the LLM's prompt, adjusting temperature, or removing the code interpreter entirely for non-algorithmic problems. This framework also allows us to formalize important properties of strategies: computational cost and intermediate traces (Appendix B.4), formal semantics (Appendix B.2), and structural characteristics (discussed in the next section).

2.2 The Strategy Landscape

We organize strategies into several categories based on their structure. A strategy is *dynamic* if it contains conditionals, meaning the control flow can change at runtime based on the input. A dynamic strategy that is recursive (uses **recfun** as shown in Figure 2a) is called an *agent*. Strategies that do not use recursion are called *workflows* since their runtime behavior is less dynamic, and a workflow that is not dynamic is called a *pipeline*. Strategies can also use parallelization to solve several tasks concurrently or to produce multiple responses at once, which can improve accuracy and reliability [11]. Table 1 shows a breakdown of five commonly used strategies based on their use of parallel compute, their dynamic or agentic nature, and their tool utilization. A description of the five strategies is in Appendix B.6. Our classification differs from common definitions that do not classify Eval-Opt as agentic and lack a clear way to determine when a strategy is agentic [12].

Figure 3 shows that different strategies result in significantly different behavior in terms of cost and accuracy. Among strategies we consider, the Code strategy is best for 3-SAT and Word Sorting, but worst for AIME and Movie Recommendation. Even among strategies with comparable accuracy, cost can vary dramatically: Eval-Opt generally achieves similar accuracy to Self-Consistency but often at half the cost. Notably, more expressive strategies do not guarantee better performance: while agentic strategies can the-

S	::=	$\mathbf{base}_P$	base process
	|	$S_1;\ S_2$	sequential composition where $S_1 : A \xRightarrow{\Sigma_1} C$ and $S_2 : C \xRightarrow{\Sigma_2} B$ and Σ_1, Σ_2 are views of Σ
	|	$S_1 \parallel S_2$	parallel composition where $S_1 : A \xRightarrow{\Sigma_1} B_1$ and $S_2 : A \xRightarrow{\Sigma_2} B_2$ and $B = (B_1, B_2)$ and Σ_1, Σ_2 are disjoint views of Σ
	|	**if** S_1 **then** S_2 **else** S_3	conditional where $S_1 : A \xRightarrow{\Sigma_1}$ Bool, $S_2 : A \xRightarrow{\Sigma_2} B$, and $S_3 : A \xRightarrow{\Sigma_3} B$ and $\Sigma_1, \Sigma_2, \Sigma_3$ are views of Σ
	|	**recfun** $f : S_1$	recursive definition of process f where $S_1 : A \xRightarrow{\Sigma_1} B$ which can internally use f and Σ_1 is a view of Σ.

(a) Grammar for strategies where $S : A \xRightarrow{\Sigma} B$.

```
1 recfun CodeAct :
2   CallLLM;
3   if ContainsCode
4   then (ExecCode; CodeAct)
5   else return
```

(b) Example: CodeAct Strategy

Figure 2: High-level grammar for strategies as compositions of base processes (left) with an example of how CodeAct is represented in this grammar (right). A further formalization is provided in Appendix B where we define how to additionally construct processes from pure functions as well as processes for explicitly accessing and updating the state, and some useful base processes such as **return**.

oretically emulate simpler strategies through their conditional and recursive structure, they may fail to select the right behavior in practice and, even when successful, often incur substantially higher computational costs than using the simpler strategy directly (for example, CodeAct is not the most effective strategy in Figure 3, despite being the most general). These observations highlight a fundamental trade-off where practitioners must balance solution quality against computational expense, often termed "overthinking."

2.3 Incorporating Feedback to Strategies

The observations above motivate systems that can adapt their strategy selection to each problem. Ideally, such systems should improve with experience: using information from past problem-solving attempts to make better strategy choices on future problems. This requires maintaining a state during inference that captures which strategies work well for which types of problems.

Existing work addresses two separate problems but not both simultaneously. Some work focuses on maintaining state during testing but is limited in what the state can influence. These methods add state to strategies compiled before inference, so they are limited to steering strategies by changing their input [6, 9]. Other work adapts general strategies to experience but requires expensive offline optimization to compile an optimized strategy that is then left stateless during inference [8, 10]. Figure 1 illustrates how training-based optimization methods (b) compare to state-based adaptation approaches (a) and our approach (c).

The success of offline strategy optimization in modifying entire strategies motivates a method that compiles useful strategies on the fly by learning from experience. This is the approach we take and introduce in the following section.

3 Experience-Guided Reasoner (EGUR)

In this section, we introduce a system that generates tailored strategies dynamically at inference time by learning from accumulated experience. Our approach centers on two key components: a *Guide* that gen-

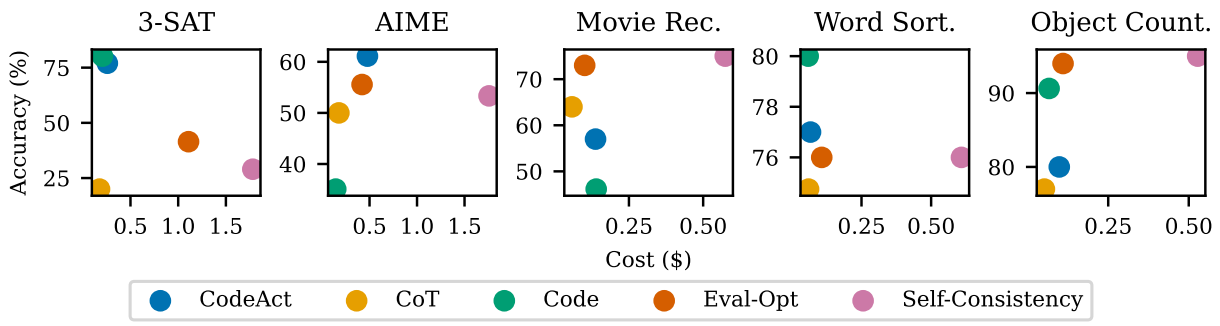

Figure 3: Comparison of strategy performance across tasks for Claude 3.7 Sonnet. Strategies closer to the top-left corner are best for the task in terms of accuracy and cost. The optimal strategy differs significantly across tasks. For example, Code excels on 3-SAT and Word Sorting but performs poorly on Movie Recommendation and AIME. Descriptions of the above strategies are provided in Appendix B.6.

Table 1: Commonly used strategies in terms of their used design patterns and tools. CI stands for code interpreter. Pipelines do not use conditionals, workflows are any strategy using conditionals, and agents are strategies using recursion.

Name	Parallelization	Conditionals	Recursion	Tools (base processes)
Pipelines	–	✗	–	–
CoT [13]				LLM
Self-Consistency [11]	✓			LLM
Workflows	–	✓	–	–
Code [14]		✓		LLM, CI
Agents	–	–	✓	–
Eval-Opt [15]		✓	✓	LLM
CodeAct [16]		✓	✓	LLM, CI

erates complete, problem-specific strategies based on past experience, and a *Consolidator* that processes execution outcomes to improve future strategy generation.

3.1 General Design

The **Guide** generates complete strategy specifications tailored to each problem based on the current context and past experiences. Rather than making decisions step-by-step like traditional agents, it produces a complete computational procedure upfront–specifying LLM calls, sampling parameters, tools, and control flow. The **Consolidator** processes execution outcomes, including reasoning traces and verifier feedback, maintaining a structured memory that improves strategy generation.

Formally, these components have types:

$$\text{Guide} : \text{Str} \overset{\Sigma}{\Longrightarrow} \left(\text{Str} \overset{\sigma}{\Longrightarrow} \text{Str}\right) \qquad \text{Consolidator} : \text{List[Exp]} \overset{\Sigma}{\Longrightarrow} \Sigma$$

where Exp represents (question, answer, trace, cost, feedback) tuples. The state space is divided into Σ (between-episode memory maintained by Consolidator) and σ (within-episode memory used during strategy execution). The Guide has a state Σ, takes a string as input and outputs a strategy as defined in Section 2.1 (which itself uses a state σ and takes a string as input and outputs a string). The Consolidator updates Σ based on execution experiences.

The complete EGUR process is shown in Algorithm 1, and we also provide an equivalent definition using our strategy language in Appendix B.5.

Algorithm 1 Experience-Guided Reasoner

Require: Question q, context Σ, and exploration factor k.
Ensure: Answer a and updated context Σ'.

$\{S_1, \ldots, S_k\} \leftarrow$ **Guide**(q, Σ, k) // generate k candidate strategies
$e = []$ // initialize experience collection
for $i = 1$ **to** k in **parallel do**
  $\sigma \leftarrow$ null // initialize strategy state
  $(a_i, t_i, c_i) \leftarrow$ **Execute**(S_i, q, σ) // execute strategy S_i
  $f_i \leftarrow \text{verifier}(q, a_i)$ // get verifier feedback
  $e \leftarrow e + [(q, a_i, t_i, c_i, f_i)]$ // collect experience tuple
end for
$\Sigma' \leftarrow$ **Consolidator**(e, Σ) // update context with experiences
$a \leftarrow a_1$ // use answer from first strategy
return a, Σ'

3.2 Strategy Selection with the Guide

The Guide generates k candidate strategies for each problem by conditioning on the query q and relevant experiences from the context Σ. This design provides some key benefits: **observability** of the system's behavior with complete strategy specifications; **exploration** of the space of possible strategies by generating multiple candidates; and **comparison** of the relative performance of different approaches for future strategy generation. The exploration factor k balances computational cost with strategy diversity—setting $k = 1$ minimizes overhead, while larger values enable discovery of better strategies at increased cost.

We invoke the Guide k times on problem q to obtain k candidate strategies $\{S_1, \ldots, S_k\}$, which we execute to obtain performance measures. Similar to Group Relative Policy Optimization (GRPO) [17], we use relative comparisons within each group of k strategies: the Consolidator records which strategies perform better relative to alternatives on similar problems, enabling the Guide to improve over time. In practice, the Guide is implemented as an LLM call that produces strategy code, which is then parsed from the response and executed (see Appendix F.1 for the prompt).

3.3 Feedback Integration with the Consolidator

The Consolidator maintains structured memory to guide future strategy generation. Rather than accumulating raw experiences, it performs selective abstraction to maximize information utility while preventing memory bloat. The Consolidator processes experiences $e = [(q_1, a_1, t_1, c_1, f_1), \ldots, (q_k, a_k, t_k, c_k, f_k)]$ from executed strategies, where each tuple contains the query, answer, execution trace, computational cost, and verifier feedback.

The context Σ maintains two key components: a **Strategy Library** that stores successful strategies with their problem characteristics for reuse and template generation, and **General Notes** that capture high-level insights about strategy effectiveness, common failure patterns, useful techniques, and problem-solving heuristics. The entire context is passed directly to the Guide for each problem, enabling it to perform retrieval and synthesis in-context rather than requiring external retrieval mechanisms.

To prevent unbounded memory growth, the Consolidator implements selective retention policies, prioritizing recent experiences and frequently accessed patterns while removing outdated information to maintain responsiveness to evolving problem distributions while preserving valuable learned knowledge. In practice, the Consolidator is implemented as an LLM call that produces memory insertions and deletions which are parsed and applied (see Appendix F.2 for the prompt).

4 Experiments

We evaluate EGuR across diverse reasoning tasks to answer four key research questions: (RQ1) Does it outperform baselines in accuracy and cost? (RQ2) Does dynamic strategy generation improve over existing

stateful methods? (RQ3) Is comparative strategy evaluation important for continual improvements? (RQ4) Does EGUR learn novel and useful strategies from experience?

4.1 Experimental Setup

Datasets. We evaluate on five datasets: AIME (2022-2024 for training, 2025 held-out), 3-SAT (5-40 variables, 400 training/40 test samples), and three BBEH [18] tasks (movie recommendation, word sorting, object counting, 100 training/100 test each). We process training data in shuffled batches of ten samples, evaluating all ten in parallel then sequentially updating based on each sample's experience.

Models. Claude 3.7 Sonnet, Qwen3-Next-80B-A3B-Thinking, and GPT-OSS-120B.

Baselines. CodeAct [16] (stateless code-interpreter agent), Dynamic Cheatsheet [6] (CodeAct with string-based memory appended to inputs), and Mem0 (CodeAct with vector database memory). We modify Dynamic Cheatsheet to incorporate verifier feedback during updates.

Verifiers. For all tasks except for 3-SAT, we use ground-truth based verifiers that compare the final answer against the correct solution. For 3-SAT, we verify the proposed assignment satisfies all clauses using a satisfiability checker. The verifier provides only binary feedback (correct/incorrect) which is passed to the Consolidator.

Implementation details. All LLM calls use temperature 0 and a maximum token budget of 20,000 tokens. For Claude 3.7 Sonnet, we enable extended thinking mode with a maximum thinking budget of 10,000 tokens; when thinking mode is enabled, Claude's temperature is automatically set to 1.0. These settings apply to all strategy executions, Guide calls, and Consolidator calls across all methods. With EGUR, the strategy can include LLM calls without thinking mode or with non-zero temperature.

Evaluation. We report prequential accuracy (accuracy before updating on each sample) and cumulative cost for strategy execution (excluding system feedback and update costs). Continual updates use one seed; held-out evaluation uses three seeds. Costs are in USD based on API pricing.

4.2 RQ1: Does EGUR outperform baselines in accuracy and cost?

Figure 4 shows EGUR's performance on held-out evaluation sets as training progresses. Two key trends emerge: accuracy increases with more experience, and cost decreases as EGUR learns more efficient strategies. Notably, Dynamic Cheatsheet's costs are significantly higher than other approaches (often exceeding $1.00 per sample) due to the increasing context length from accumulating memory, and the underlying strategy taking many turns before outputting a final answer. In contrast, EGUR maintains low and decreasing costs through selective memory management. Mem0 shows minimal cost reduction and modest accuracy improvements, suggesting limited adaptation capability.

Table 3 provides detailed prequential accuracy and cost results across all datasets and models. EGUR consistently achieves the best accuracy-cost trade-offs: on Claude 3.7 Sonnet, EGUR-5 achieves 96.0% accuracy on 3-SAT at $0.152 cost versus CodeAct's 77.0% at $0.257 and Dynamic Cheatsheet's 89.9% at $76.353. Similar patterns hold across models and tasks.

4.3 RQ2: Does dynamic strategy generation improve over existing stateful strategies?

The key distinction between EGUR and existing stateful methods is architectural: **EGUR generates a complete strategy specification for each problem before execution**, while existing methods **steer dynamic strategies at runtime by modifying their inputs**. Concretely, Mem0 and Dynamic Cheatsheet prepend memory to a fixed CodeAct agent's inputs, influencing its behavior indirectly. In contrast, EGUR produces entirely new strategy specifications—including prompts, sampling parameters (temperature, max tokens), tool availability, and control flow—tailored to each problem.

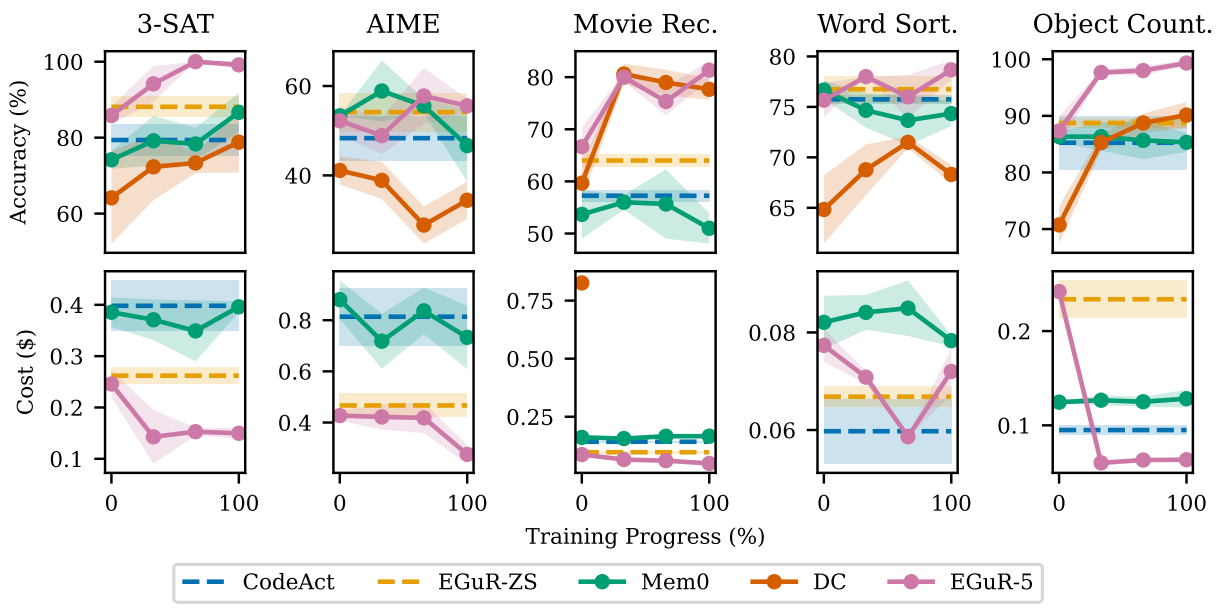

Figure 4: Evolution of accuracy and cost on held-out evaluation sets as training progresses for Claude 3.7 Sonnet. EGUR-5 consistently improves accuracy while reducing cost with more experience. Cost is shown up to $1.0 for visualization; Dynamic Cheatsheet (DC) typically exceeds this threshold, reaching $9.95, $2.26, $2.88, $4.32, and $7.16 per sample after training on 3-SAT, AIME, Movie Rec., Word Sort., and Object Count., respectively. The complete results for Claude 3.7 Sonnet, GPT-OSS-120B, and Qwen3-Next-80B-A3B-Thinking are included in Appendix C.

This architectural difference has profound implications. Methods that steer via input modification cannot remove tools, change sampling parameters, or switch between agentic and workflow paradigms. EGUR is significantly more flexible: for instance, it learns to completely remove the code interpreter tool when it harms performance, adjust temperature from 0.7 to 0.0 for deterministic tasks, and replace multi-turn agentic strategies with single-call workflows when appropriate.

The results in Figure 4 reflect these differences. Mem0 achieves minimal cost reduction despite maintaining memory, as it cannot fundamentally change the underlying strategy's computational structure. Dynamic Cheatsheet's costs grow unboundedly as accumulated memory increases context length in every call. EGUR achieves superior accuracy while reducing costs by 2-111× through complete strategy adaptation.

4.4 RQ3: Is comparative strategy evaluation important for continual improvements?

We ablate the exploration level k (strategies generated per problem) to assess the importance of comparative evaluation. Figure 5 shows results for Claude 3.7 Sonnet. EGUR-ZS (zero-shot, no memory updates) serves as a stateless baseline, EGUR-1 generates one strategy per problem and learns from absolute feedback, while EGUR-5 generates five strategies enabling comparative evaluation.

Three findings emerge. First, memory is essential: EGUR-1 consistently outperforms EGUR-ZS, showing that learning from experience helps even without comparison. Second, comparative evaluation provides substantial additional benefits: EGUR-5 outperforms EGUR-1 on most tasks, with large improvements on 3-SAT and object counting. Third, cost decreases with higher k because comparative evaluation helps discover efficient strategies faster.

4.5 RQ4: Does EGUR learn novel and useful strategies from experience?

We analyze strategies generated before and after training to identify learned adaptations. EGUR exhibits several consistent patterns: for CodeAct strategies, it learns to specify allowed libraries, include useful code

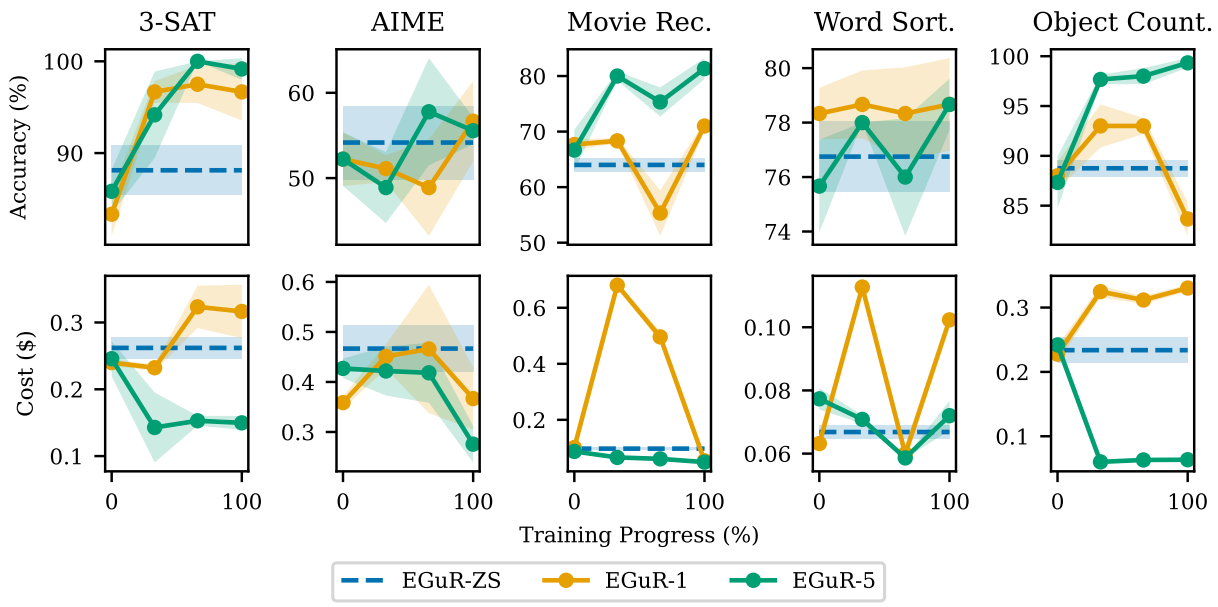

Figure 5: Ablation of exploration level in EGuR for Claude 3.7 Sonnet. Higher exploration levels (more strategies per problem) generally improve both accuracy and cost-efficiency by enabling comparative evaluation of strategy effectiveness.

snippets, and add error handling; more generally, it increases specificity when general approaches fail but simplifies when appropriate.

We also find that the code interpreter tool sometimes harms performance while increasing cost. For BBEH object counting, intuition suggests CodeAct would excel (problems involve adding many numbers), but EGuR converges on a single LLM call with detailed instructions which proves more accurate and substantially cheaper. The learned strategy includes specific guidance for parsing text, categorizing items, and handling quantity changes. The strategy is shown in Appendix D.

Similarly, for BBEH word sorting, EGuR learns to distinguish algorithmic sorting problems (which benefit from Python's `sort`) from reasoning problems (identifying logical mistakes in explanations). For the latter, it adopts CoT with verifier-based fallback rather than CodeAct.

These findings show EGuR learns meaningful heuristics for when to use tools versus LLM-based reasoning, when to increase versus decrease computational investment, and how to tailor strategies through prompts to problem characteristics. Additional examples appear in Appendix D.

5 Related Work

Prompting strategies. Prompting methods including Chain-of-Thought (CoT) [13], Self-Consistency [11], Program-of-Thoughts (PoT) [14], and many other approaches [19–22] elicit intermediate reasoning traces but rely on fixed, non-adaptive scaffolds. More recent approaches like CodeAct [16], Self-Discover [6], and Meta-Prompting [23] dynamically compose solution solving methods per instance, but still do not leverage past experience or adapt their strategies over time. These methods are thus dynamic at inference but remain stateless across instances.

Prompt adaptation from inference-time state or training. Another line of work adapts prompts or parameters within fixed agent workflows. DSPy [2] and TextGrad [5] are systems to optimize prompts or even the inputs to fixed strategy structures using a training dataset with labels or an answer verifier, while Dynamic Cheatsheet [6] and Buffer of Thoughts [7] accumulate and retrieve artifacts during inference for instance-level adaptation. Concurrent to this work, training-free GRPO [24] was proposed as an effective

method for online prompt optimization which is similar to our GRPO inspired approach to producing effective memory updates from the Consolidator, and Agentic Context Engineering [25] improves over Dynamic Cheatsheet by providing more structure for memory updates. Systems such as DSPy are highly general and allow for many different algorithms for offline prompt optimization [26–28]. Other methods such as Per-Instance Program Synthesis (PIPS) [29], model routing [30–32], and proprietary products like GPT-5 [33] rely on trained routers to select between a fixed set of strategies on a per-instance basis. These approaches achieve stronger performance through experience but are limited in their adaptability based on only modifying the input to a fixed strategy or selecting between a fixed number of strategies.

Full strategy adaptation. Beyond prompt tuning, ADAS [8] treats agent design as a search problem over Python programs defining agents, enabling strategy-level adaptation. However, ADAS and similar approaches [34–37] operate offline and at the task level, requiring multiple episodes to converge. In contrast, our method introduces an *online adaptive meta-agent* that performs dynamic, instance-level adaptation and test-time strategy search, combining the benefits of prior work on dynamic strategies, memory-based learners, and automated design in a unified framework.

6 Conclusion

We have described a system, EGUR, which generates complete strategies—computational procedures involving LLM calls, tools, and control logic—dynamically at inference time based on accumulated experience. Specifically, a Guide component generates problem-specific strategy specifications conditioned on past experience, while a Consolidator maintains structured memory of strategy effectiveness. Unlike existing adaptive systems that use memory for textual steering of an LLM or agent, our approach has much greater flexibility, enabling adaptation of all strategy components (prompts, sampling parameters, tool configurations, and control logic). Across five benchmarks, EGUR achieves up to 14% relative accuracy improvements while reducing computational costs by up to 111x, with both metrics improving as the system gains experience. Our analysis reveals EGUR learns meaningful heuristics: when to deploy expensive agentic strategies versus lightweight workflows, when tools help versus harm, and how to tailor computational investment to problem difficulty.

Our approach has some limitations that suggest directions for future work. First, EGUR uses ground truth feedback from verifiers to learn effectively. Exploring whether this feedback can be replaced with weaker signals—such as LLM-based evaluation—is an important direction. Second, the effectiveness of the system hinges on the Guide's zero-shot strategy generation capabilities, which may be suboptimal for unfamiliar problem types. In this case, it may be necessary to train or optimize the Guide through reinforcement learning or other training methods. Similarly, the Consolidator relies on an LLM to manage memory, which may not optimally balance memory size against information utility, and may benefit from meta-learning approaches for more effective memory management.

References

[1] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

[2] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=sY5N0zY5Od`.

[3] Yichen Huang and Lin F Yang. Gemini 2.5 pro capable of winning gold at imo 2025. *arXiv preprint arXiv:2507.15855*, 2025.

[4] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

[5] Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

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

[7] Ling Yang, Zhaochen Yu, Tianjun Zhang, Shiyi Cao, Minkai Xu, Wentao Zhang, Joseph E Gonzalez, and Bin Cui. Buffer of thoughts: Thought-augmented reasoning with large language models. *Advances in Neural Information Processing Systems*, 37:113519–113544, 2024.

[8] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

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

[10] Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms. *Advances in Neural Information Processing Systems*, 37:71596–71642, 2024.

[11] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

[12] Erik Schluntz and Barry Zhang. Building effective ai agents. https://www.anthropic.com/engineering/building-effective-agents, December 2024. Accessed: 2025-09-30.

[13] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[14] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=YfZ4ZPt8zd.

[15] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

[16] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024.

[17] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[18] Mehran Kazemi, Bahare Fatemi, Hritik Bansal, John Palowitch, Chrysovalantis Anastasiou, Sanket Vaibhav Mehta, Lalit K Jain, Virginia Aglietti, Disha Jindal, Peter Chen, et al. Big-bench extra hard. *arXiv preprint arXiv:2502.19187*, 2025.

[19] Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. Faithful chain-of-thought reasoning. In *The 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (IJCNLP-AACL 2023)*, 2023.

[20] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

[21] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, and Ed H. Chi. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WZH7099tgfM.

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

[23] Mirac Suzgun and Adam Tauman Kalai. Meta-prompting: Enhancing language models with task-agnostic scaffolding. *arXiv preprint arXiv:2401.12954*, 2024.

[24] Yuzheng Cai, Siqi Cai, Yuchen Shi, Zihan Xu, Lichao Chen, Yulei Qin, Xiaoyu Tan, Gang Li, Zongyi Li, Haojia Lin, et al. Training-free group relative policy optimization. *arXiv preprint arXiv:2510.08191*, 2025.

[25] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025.

[26] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

[27] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.

[28] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=92gvk82DE-.

[29] Adam Stein, Neelay Velingker, Mayur Naik, and Eric Wong. Once upon an input: Reasoning via per-instance program synthesis. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=XsQNqRdcdh.

[30] Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=cSimKw5p6R.

[31] Ruochen Wang, Sohyun An, Minhao Cheng, Tianyi Zhou, Sung Ju Hwang, and Cho-Jui Hsieh. One prompt is not enough: Automated construction of a mixture-of-expert prompts. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=edHLN4ODWu.

[32] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs from preference data. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=8sSqNntaMr.

[33] OpenAI. Introducing gpt-5. https://openai.com/index/introducing-gpt-5/, Aug 2025.

[34] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=z5uVAKwmjf.

Table 2: Taxonomy of related work by **S@T**=Stateful at test, **Adapted**=What is adapted in response to the state, input, or training samples, **Training-free**=If the method does not require an initial training stage

Method	S@T	Adapted	Training-free
Stateless strategies (human designed)			
CoT			✓
Self-Consistency			✓
PoT (non-iterative)			✓
CodeAct [16]			✓
Adapts prompts from state/training			
Self-Discover [6]		Prompt	✓
DSPy [2]		Prompts	
TextGrad [5]		Prompts, Inputs	
Buffer of Thoughts [7]	✓	Prompt	✓
Dynamic Cheatsheet [6]	✓	Prompt	✓
Full strategy adaptation			
FlowReasoner [38]		Strategy	
ADAS [8]		Strategy	
EGUR	✓	Strategy	✓

[35] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024. URL `https://openreview.net/forum?id=uTC9AFXIhg`.

[36] Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. Autoflow: Automated workflow generation for large language model agents. *arXiv preprint arXiv:2407.12821*, 2024.

[37] Guibin Zhang, Kaijie Chen, Guancheng Wan, Heng Chang, Hong Cheng, Kun Wang, Shuyue Hu, and Lei Bai. Evoflow: Evolving diverse agentic workflows on the fly. *arXiv preprint arXiv:2502.07373*, 2025.

[38] Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. Flowreasoner: Reinforcing query-level meta-agents. *arXiv preprint arXiv:2504.15257*, 2025.

A Related Work Taxonomy

A taxonomy of the related work is shown in Table 2.

B Strategies

B.1 Syntax

The syntax of strategies is separated into values, expressions and programs.

Values

$$
\begin{array}{rcl}
v & ::= & \text{str} \mid \text{float} \mid \text{true} \mid \text{false} \mid \text{null} \\
& \mid & [v \ (,\ v)*] \\
& \mid & \{\text{str} : v \ (,\ \text{str} : v)*\}
\end{array}
$$

Values are the typical Python values including strings, floats, booleans, the None value (written null) as well as lists and dictionaries.

Expressions

$$\begin{array}{rcl} e & ::= & x \mid v \mid e[e] \mid e +\!\!+ e \mid e \oplus e \mid e + e \mid e - e \mid e * e \mid e \div e \\ & \mid & \textbf{lambda}\ x.\ e \mid e(e) \end{array}$$

Expressions consist of either a variable x, a value, or various operations of expressions including list indexing, list append $+\!\!+$, dictionary append $\oplus$, float arithmethic $(+, -, *, \div)$, lambda functions, and function application.

Programs

$$\begin{array}{rcl} p & ::= & \text{base\_proc} \mid \textbf{return} \mid \textbf{pure}\ e \mid \textbf{get} \mid \textbf{put}\ e \\ & \mid & p; p \\ & \mid & p \parallel p \\ & \mid & \textbf{if}\ p\ \textbf{then}\ p\ \textbf{else}\ p \\ & \mid & \textbf{fix}\ p.\ p \end{array}$$

Programs are then either a base process, a return statement (which directly returns the input), an expression lifted to a program using **pure**, a **get** which returns the current state of the program, a **put** of an expression which updates the state with the expression, or a sequential or parallel composition of programs, a conditional of programs, or a fixpoint of a program.

B.2 Semantics

We now provide a denotational semantics for the strategy language. First, a program in this language is a *stateful process* mapping between two sets:

$$\text{Proc}(A, B, \Sigma) : (A \times \Sigma) \to (B \times \Sigma)$$

where Σ is the state space of the process. Intuitively, a process can be thought of as a mapping from A to B which has access to some state Σ. To simplify how we compose processes, we use the state monad:

$$\text{State}(A) : \Sigma \to (A \times \Sigma)$$

and then rewrite the process type as $\text{Proc}(A, B, \Sigma) : A \to \text{State}(B)$ which can be thought of as a function which maps an input in A to a computation which when given a state will produce an output in B and an updated state. The choice of the state space Σ determines how processes share information when composed as well as allows for tracking information such as execution traces, cost, and strategy structure. For instance, we use $\Sigma = M \times P \times \text{Trace} \times \text{Cost}$ where M is the current conversation log of any model calls, P is the execution state for any code execution, Trace is the current execution trace, and Cost is the current execution cost of the strategy.

The program meaning function $[\![\cdot]\!]$ is now a mapping from the syntactic forms defined above to processes.

$$[\![\cdot]\!] : \mathsf{Program} \longrightarrow \text{Proc}(A, B, \Sigma)$$

$$
\begin{aligned}
[\![\text{base\_proc}]\!] &= \text{base\_proc}, \\
[\![\textbf{return}]\!] &= \lambda\, a.\, \lambda c.\, (a, c), \\
[\![\textbf{pure}\ f]\!] &= \lambda\, a.\, \lambda c.\, ([\![f]\!](a), c), \\
[\![\textbf{get}]\!] &= \lambda\, a.\, \lambda c.\, (c, c), \\
[\![\textbf{put}\ e]\!] &= \lambda\, a.\, \lambda c.\, (a, [\![e]\!]), \\
[\![p_1; p_2]\!] &= \lambda\, a.\, \lambda c.\, \text{let } (b, c_1) \leftarrow [\![p_1]\!](a)(c) \\
&\quad\ \text{in } [\![p_2]\!](b)(c_1), \\
[\![p_1 \parallel p_2]\!](a)(c) &= \lambda\, a.\, \lambda c.\, \text{let } (b_1, c_1) \leftarrow [\![p_1]\!](a)(c) \text{ and } (b_2, c_2) \leftarrow [\![p_2]\!](a)(c), \\
&\quad\ \text{in } ((b_1, b_2), (c_1, c_2)), \\
[\![\textbf{if}\ p_1\ \textbf{then}\ p_2\ \textbf{else}\ p_3]\!](s) &= \lambda\, a.\, \lambda c.\, \text{let } (b, c_1) \leftarrow [\![p_1]\!](a)(c) \\
&\quad\ \text{in } \begin{cases} [\![p_2]\!](a, c_1), & \text{if } b = \mathsf{true} \\ [\![p_3]\!](a, c_1), & \text{if } b = \mathsf{false} \end{cases}, \\
[\![\textbf{fix}\, p_1.\, p_2]\!] &= \mathrm{lfp}_{\sqsubseteq}\, \big(p \mapsto [\![p_2]\!]_{p_1 \mapsto p}\big).
\end{aligned}
$$

B.3 Guide and Consolidator

The Guide is a strategy which takes a query as input and outputs a strategy for solving the problem, as well as has access to a state which it can use to store information between queries. Therefore, the Guide is just a higher-order strategy, or a stateful mapping from inputs to strategies and its type is the following:

$$\mathrm{Guide}(A, B, \Sigma) : \mathrm{Proc}(A, \mathrm{Proc}(A, B, \sigma), \Sigma).$$

The memory updater can also be written as a strategy which takes the current experience, the current running state, and outputs the updated state:

$$\mathrm{Consolidator}(\mathrm{Exp}, \Sigma) : \mathrm{Proc}(\mathrm{Exp}, \Sigma, \Sigma).$$

B.4 Strategy Cost

We define the cost of a strategy recursively based on the costs of the base processes $P \in \mathcal{P}$ defined by $c_P(s) \in \mathbb{R}_{\geq 0}$, the cost of running P on input s.

$$
\begin{aligned}
&\mathrm{Cost} : \mathsf{Term} \times S \longrightarrow \mathbb{R}_{\geq 0} \cup \{\infty\}, \\
&\mathrm{Cost}(t, s) = \begin{cases}
c_P(s), & \text{if } t = P \in \mathcal{P}, \\
\mathrm{Cost}(t_1, s) + \mathbb{E}_{s' \sim [\![t_1]\!](s)}\big[\mathrm{Cost}(t_2, s')\big], & \text{if } t = t_1; t_2, \\
\mathrm{Cost}(t_1, s) + \mathrm{Cost}(t_2, s), & \text{if } t = t_1 \parallel t_2, \\
\begin{cases} \mathrm{Cost}(t_1, s), & b(s) = \mathsf{true}, \\ \mathrm{Cost}(t_2, s), & b(s) = \mathsf{false}, \end{cases} & \text{if } t = \text{if } b \text{ then } t_1 \text{ else } t_2, \\
\mathrm{Cost}\big(t[X \mapsto \textbf{fix}\, X.\, t], s\big), & \text{if } t = \textbf{fix}\, X.\, t \quad \text{(least fixed point)}.
\end{cases}
\end{aligned}
$$

B.5 EGUR Implemented as a Strategy

We show how EGUR is represented as a strategy in Algorithm 2 where we make sequential and parallel composition of processes expliclit.

Algorithm 2 Experience-Guided Reasoner

Require: Question q, context Σ, and exploration factor k.
Ensure: Answer a and updated context Σ'.

update (**fun** (q, Σ): $\Sigma + \{\text{“question”} : q\}$); // store the question in context
Guide$_k$; // generate k strategies conditioned on the question and context
(**recfun** ExecuteStrategies: // execute all strategies in parallel
    if *IsEmpty*
    then return
    else (*GetFirst*; **Trace**; *Verify*) || (*PopFirst*; ExecuteStrategies)
);
update (**fun** (e, Σ): $\Sigma + \{\text{“answer”} : e[0][0]\}$); // store the answer from the first strategy
Consolidator // update the context from the k experiences
get “answer”

```
CallLLM
```
(a) CoT

```
(CallLLM || CallLLM ||
 CallLLM || CallLLM ||
 CallLLM);
MajorityVote
```
(b) Self-Consistency

```
CallLLM;
(if ContainsCode
 then ExecCode
 else return)
```
(c) Code

```
recfun Eval-Opt :
  CallOptLLM;
  (if EvalLLM
   then return
   else Eval-Opt)
```
(d) Eval-Opt

Figure 6: Implementation of CoT, Self-Consistency, Code, and Eval-Opt as strategies

B.6 Common Strategies

In this section, we describe the five strategies shown in Table 1. We also show the implementations of the strategies in our strategy language in Figure 6.

The CoT strategy is a single LLM inference call without any tools, the Code strategy is a single round of code writing and executing the code with an interpreter to produce the answer, the Eval-Opt strategy is an evaluator-optimizer loop where the optimizer produces candidate solutions and the evaluator critiques the solutions, the Self-Consistency strategy samples 10 solutions from the CoT strategy and then returns the majority answer, and the CodeAct strategy uses multiple rounds of code generation and execution to solve the problem, and is prototypical of an *agentic* workflow.

C Full results

The prequential evaluation results for all models is included in Table 3.

Full results for all methods on all three models and five holdout datasets across four evenly spaced checkpoints during training are shown in Table 4, Table 5, Table 6, Table 7, and Table 8.

D Example Generated Strategies

The following shows an example of a strategy generated with GPT-OSS-120B to solve a sarcasm task (Sarc Triples) from BBEH. We can see that this strategy includes a prompt tailored to the specific problem, majority voting, and modification of the temperature parameter to the called model.

Generated Strategy

```
def strategy(x):
    """
    Detect sarcasm in three Reddit replies.
```

Table 3: Prequential Accuracy (%) and Cost ($) on each dataset. EGuR consistently achieves the best accuracy-cost trade-offs, with bold indicating best accuracy and underline indicating second-best. Ours-k denotes EGuR with exploration level k.

Model	Method	3-SAT		AIME		Movie Rec.		Word Sort		Object Count	
		Acc	Cost	Acc	Cost	Acc	Cost	Acc	Cost	Acc	Cost
Claude	CodeAct	0.770	0.257	**0.611**	0.478	0.570	0.137	0.770	**0.067**	0.800	0.098
	CodeAct+Mem0	0.860	0.325	0.589	0.548	0.540	0.141	0.760	0.085	0.870	0.174
	DynamicCheatsheet	0.899	76.353	0.400	8.615	**0.840**	9.152	0.700	8.231	0.870	24.895
	EGuR-5	**0.960**	**0.152**	0.589	**0.292**	0.700	**0.067**	**0.800**	0.073	**0.960**	**0.075**
Qwen3	CodeAct	**0.875**	0.032	0.622	0.010	0.680	0.002	**0.700**	0.003	0.400	0.017
	CodeAct+Mem0	0.875	0.037	0.656	0.011	**0.750**	0.003	0.680	0.004	0.360	0.019
	DynamicCheatsheet	0.125	0.570	0.467	0.124	0.550	0.205	0.450	0.171	0.180	0.469
	EGuR-3	0.695	**0.005**	**0.767**	**0.006**	0.650	**0.002**	0.650	**0.001**	**0.540**	**0.005**
GPT	CodeAct	**0.655**	0.236	0.778	**0.073**	**0.550**	0.091	**0.780**	**0.087**	0.690	**0.047**
	CodeAct+Mem0	0.640	0.219	**0.800**	0.078	0.530	**0.061**	0.780	0.129	**0.790**	0.059
	EGuR-3	0.650	**0.100**	0.756	0.095	0.350	0.064	0.580	0.091	0.620	0.090

Table 4: Accuracy (%) across training progress for different methods on 3-SAT and AIME datasets. Values shown as mean ± standard deviation across seeds. Claude uses EGuR-5, GPT and Qwen use EGuR-3. **Bold** indicates best performance within each model, underlined indicates second best within each model.

Model	Method	3-SAT				AIME			
		0%	33%	66%	100%	0%	33%	66%	100%
Claude-3.5-Sonnet	CodeAct	79.4±4.1	79.4±4.1	79.4±4.1	79.4±4.1	48.3±5.0	48.3±5.0	48.3±5.0	48.3±5.0
	EGuR-ZS	**88.1±2.7**	88.1±2.7	88.1±2.7	88.1±2.7	**54.2±4.3**	54.2±4.3	54.2±4.3	54.2±4.3
	Mem0	74.2±2.4	79.2±6.6	78.3±4.2	86.7±5.1	53.3±0.0	**58.9±6.8**	55.6±1.6	46.7±8.2
	DC	64.1±12.1	72.3±8.8	73.3±2.7	78.8±8.0	41.1±3.1	38.9±4.2	28.9±4.2	34.4±4.2
	EGuR-5	85.8±2.4	**94.2±4.7**	**100.0±0.0**	**99.2±1.2**	52.2±3.1	48.9±4.2	**57.8±6.3**	**55.6±1.6**
GPT-OSS-120B	CodeAct	88.8±5.7	**88.8±5.7**	88.8±5.7	**88.8±5.7**	**79.6±6.8**	**79.6±6.8**	**79.6±6.8**	**79.6±6.8**
	EGuR-ZS	83.1±2.7	83.1±2.7	83.1±2.7	83.1±2.7	65.8±3.6	65.8±3.6	65.8±3.6	65.8±3.6
	Mem0	**91.7±1.2**	86.7±5.9	**91.7±2.4**	84.2±2.4	76.6±3.4	60.0±6.7	69.1±9.1	79.4±16.1
	DC	10.8±6.6	6.7±2.4	19.2±4.2	13.3±4.2	45.6±4.2	44.4±5.7	53.3±7.2	44.4±4.2
	EGuR-3	85.8±2.4	80.8±9.6	35.0±6.1	69.2±12.3	65.6±8.7	75.6±4.2	73.3±7.2	76.7±4.7
Qwen3-Next-80B	CodeAct	63.8±7.4	63.8±7.4	63.8±7.4	63.8±7.4	74.2±4.9	74.2±4.9	74.2±4.9	74.2±4.9
	EGuR-ZS	**80.0±4.7**	80.0±4.7	**80.0±4.7**	**80.0±4.7**	**78.8±7.5**	**78.8±7.5**	78.8±7.5	**78.8±7.5**
	Mem0	75.0±0.0	52.5±0.0	57.5±0.0	67.5±0.0	66.7±0.0	76.7±0.0	**80.0±0.0**	76.7±0.0
	EGuR-3	72.5±0.0	**90.0±0.0**	75.0±0.0	45.0±0.0	56.7±0.0	72.4±0.0	60.0±0.0	55.2±0.0

```
Repeatedly query the model, extract the list after "FINAL ANSWER:",
and return the majority-voted label string.
"""
import re
from collections import Counter

# Prompt that enforces step-by-step reasoning and a strict final answer format
base_prompt = (
"You are given three (post, reply) pairs. For each reply decide whether it is "
"sarcastic (1) or not sarcastic (0). Think carefully about each pair, then output "
"the three labels in order as a comma-separated list. Write the list after a line "
"that starts exactly with 'FINAL ANSWER:'.\n"
f"Problem: {x}"
)

collected_answers = []

# Number of independent attempts (adjustable; 5 balances cost and reliability)
for _ in range(5):
  # First generation
  out = M(base_prompt, temperature=0.5, max_tokens=2000)[0]

  # Try to extract the answer
  ans = extract_final_answer(out)
  if not ans:
    # Ask the model to repeat with the required format
```

Table 5: Accuracy (%) across training progress for different methods on Movie Recommendation, Word Sorting, and Object Counting datasets. Values shown as mean ± standard deviation across seeds. Claude uses EGuR-5, GPT and Qwen use EGuR-3. **Bold** indicates best performance within each model, underlined indicates second best within each model.

Model	Method	Movie Rec.				Word Sort.				Object Count.			
		0%	33%	66%	100%	0%	33%	66%	100%	0%	33%	66%	100%
Claude-3.5-Sonnet	CodeAct	57.2±1.1	57.2±1.1	57.2±1.1	57.2±1.1	75.8±0.4	75.8±0.4	75.8±0.4	75.8±0.4	85.2±4.8	85.2±4.8	85.2±4.8	85.2±4.8
	EGuR-ZS	64.0±1.2	64.0±1.2	64.0±1.2	64.0±1.2	**76.8±1.3**	76.8±1.3	**76.8±1.3**	76.8±1.3	**88.8±0.8**	88.8±0.8	88.8±0.8	88.8±0.8
	Mem0	53.7±4.6	56.0±1.4	55.7±6.6	51.0±2.9	76.7±0.9	74.7±1.7	73.7±2.5	74.3±1.2	86.3±0.9	86.3±2.5	85.7±3.3	85.3±1.7
	DC	59.7±3.1	**80.6±2.0**	**79.0±2.5**	77.7±2.1	64.8±3.4	68.8±2.5	71.5±0.6	68.3±0.9	70.7±3.3	85.3±1.5	88.7±1.7	90.1±2.3
	EGuR-5	**66.7±3.7**	80.0±0.8	75.3±2.6	**81.3±1.9**	75.7±1.7	**78.0±0.0**	76.0±2.2	**78.7±0.9**	87.3±2.6	**97.7±0.5**	**98.0±0.8**	**99.3±0.5**
GPT-OSS-120B	CodeAct	72.8±3.3	72.8±3.3	72.8±3.3	72.8±3.3	**66.0±1.7**	**66.0±1.7**	**66.0±1.7**	**66.0±1.7**	**39.5±7.1**	39.5±7.1	39.5±7.1	**39.5±7.1**
	EGuR-ZS	**75.0±3.2**	**75.0±3.2**	**75.0±3.2**	**75.0±3.2**	64.8±4.0	64.8±4.0	64.8±4.0	64.8±4.0	29.5±1.8	29.5±1.8	29.5±1.8	29.5±1.8
	Mem0	73.3±0.9	72.0±2.9	72.0±1.6	71.0±2.4	64.0±3.6	63.3±2.4	63.7±2.1	63.7±1.7	35.3±2.5	41.0±5.4	41.0±2.9	38.0±2.2
	DC	73.7±1.9	57.0±1.4	52.3±0.5	59.7±0.5	47.3±0.5	37.7±4.5	45.0±2.4	39.3±2.5	13.0±4.5	15.7±1.7	21.7±2.1	8.7±5.2
	EGuR-3	62.3±3.4	60.3±1.7	62.3±3.1	74.7±2.5	63.3±2.6	63.0±2.8	65.0±1.6	64.0±3.6	37.3±2.6	**48.3±4.8**	**57.3±1.2**	39.0±4.5
Qwen3-Next-80B	CodeAct	46.5±1.5	**46.5±1.5**	46.5±1.5	**46.5±1.5**	75.5±2.1	**75.5±2.1**	75.5±2.1	75.5±2.1	**75.5±3.2**	**75.5±3.2**	**75.5±3.2**	75.5±3.2
	EGuR-ZS	39.2±1.7	39.2±1.7	39.2±1.7	39.2±1.7	64.4±2.3	64.4±2.3	64.4±2.3	64.4±2.3	74.9±4.5	74.9±4.5	74.9±4.5	74.9±4.5
	Mem0	**47.0±0.0**	45.0±0.0	**47.0±0.0**	43.0±0.0	**77.0±0.0**	75.0±0.0	**77.0±0.0**	**78.0±0.0**	74.0±0.0	63.0±0.0	70.0±0.0	**77.0±0.0**
	EGuR-3	41.0±0.0	32.0±0.0	28.0±0.0	40.0±0.0	65.0±0.0	43.0±0.0	68.0±0.0	64.0±0.0	60.0±0.0	44.0±0.0	38.0±0.0	37.0±0.0

Table 6: Cost ($) across training progress for different methods on 3-SAT and AIME datasets. Values shown as mean ± standard deviation across seeds. Claude uses EGuR-5, GPT and Qwen use EGuR-3. **Bold** indicates lowest cost within each model, underlined indicates second lowest within each model.

Model	Method	3-SAT				AIME			
		0%	33%	66%	100%	0%	33%	66%	100%
Claude-3.5-Sonnet	CodeAct	0.398±0.049	0.398±0.049	0.398±0.049	0.398±0.049	0.814±0.112	0.814±0.112	0.814±0.112	0.814±0.112
	EGuR-ZS	0.262±0.016	0.262±0.016	0.262±0.016	0.262±0.016	0.467±0.046	0.467±0.046	0.467±0.046	0.467±0.046
	Mem0	0.386±0.029	0.371±0.040	0.349±0.059	0.396±0.011	0.880±0.075	0.718±0.107	0.837±0.091	0.733±0.124
	DC	8.969±1.020	10.056±1.431	10.572±1.095	9.951±1.944	1.133±0.149	1.371±0.395	2.146±0.109	2.259±0.280
	EGuR-5	**0.246±0.026**	**0.143±0.052**	**0.153±0.007**	**0.150±0.010**	**0.427±0.020**	**0.422±0.049**	**0.418±0.060**	**0.276±0.037**
GPT-OSS-120B	CodeAct	0.036±0.007	0.036±0.007	0.036±0.007	0.036±0.007	0.004±0.000	**0.004±0.000**	0.004±0.000	0.004±0.000
	EGuR-ZS	0.005±0.001	0.005±0.001	0.005±0.001	0.005±0.001	0.004±0.001	0.004±0.001	0.004±0.001	0.004±0.001
	Mem0	0.027±0.002	0.031±0.005	0.038±0.009	0.035±0.004	0.006±0.002	0.010±0.001	0.007±0.001	0.011±0.003
	DC	0.019±0.002	0.014±0.001	0.017±0.001	0.021±0.002	0.018±0.000	0.017±0.001	0.028±0.001	0.032±0.001
	EGuR-3	**0.004±0.000**	**0.003±0.000**	**0.004±0.001**	**0.004±0.001**	**0.003±0.000**	0.006±0.002	**0.004±0.001**	**0.003±0.001**
Qwen3-Next-80B	CodeAct	0.251±0.024	0.251±0.024	0.251±0.024	0.251±0.024	0.104±0.015	0.104±0.015	0.104±0.015	0.104±0.015
	EGuR-ZS	0.137±0.013	0.137±0.013	0.137±0.013	0.137±0.013	0.120±0.023	0.120±0.023	0.120±0.023	0.120±0.023
	Mem0	0.212±0.000	0.208±0.000	0.223±0.000	0.228±0.000	0.105±0.000	0.115±0.000	0.080±0.000	0.089±0.000
	EGuR-3	**0.089±0.000**	**0.097±0.000**	**0.135±0.000**	**0.096±0.000**	**0.057±0.000**	**0.038±0.000**	**0.041±0.000**	**0.054±0.000**

```
        clarification = (
          "Please respond again, making sure to include a line that starts exactly with "
          "'FINAL ANSWER:' followed by the three labels separated by commas."
        )
        out = M(clarification + "\nPrevious output:\n" + out,
                temperature=0, max_tokens=2000)[0]
        ans = extract_final_answer(out)

      if ans:
        # Normalise whitespace and store
        collected_answers.append(ans.strip())

  if not collected_answers:
    return "No answer found."

  # Majority vote over the collected answer strings
  most_common = Counter(collected_answers).most_common(1)[0][0]
  return most_common
```

For Claude 3.7 Sonnet, we show generated strategies before and after training on the BBEH Object Counting task. The first strategy below, from before training answers the particular problem it was generated for incorrectly and cost $0.166 while the following strategy answers the same problem correctly and only costs $0.062.

Table 7: Cost ($) across training progress for different methods on Movie Recommendation and Word Sorting datasets. Values shown as mean ± standard deviation across seeds. Claude uses EGuR-5, GPT and Qwen use EGuR-3. **Bold** indicates lowest cost within each model, underlined indicates second lowest within each model.

Model	Method	Movie Rec.				Word Sort.			
		0%	33%	66%	100%	0%	33%	66%	100%
Claude-3.5-Sonnet	CodeAct	0.142±0.004	0.142±0.004	0.142±0.004	0.142±0.004	**0.060±0.007**	**0.060±0.007**	0.060±0.007	**0.060±0.007**
	EGuR-ZS	0.097±0.005	0.097±0.005	0.097±0.005	0.097±0.005	0.067±0.002	0.067±0.002	0.067±0.002	0.067±0.002
	Mem0	0.161±0.004	0.156±0.005	0.166±0.007	0.166±0.005	0.082±0.005	0.084±0.004	0.085±0.006	0.078±0.001
	DC	0.826±0.011	2.525±0.047	2.710±0.034	2.885±0.014	1.214±0.047	2.822±0.133	5.426±0.066	4.322±0.068
	EGuR-5	**0.087±0.001**	**0.066±0.002**	**0.061±0.001**	**0.049±0.001**	0.077±0.003	0.071±0.001	**0.059±0.000**	0.072±0.005
GPT-OSS-120B	CodeAct	0.003±0.000	0.003±0.000	0.003±0.000	0.003±0.000	0.003±0.000	0.003±0.000	0.003±0.000	0.003±0.000
	EGuR-ZS	0.002±0.000	0.002±0.000	0.002±0.000	**0.002±0.000**	**0.001±0.000**	**0.001±0.000**	**0.001±0.000**	0.001±0.000
	Mem0	0.003±0.000	0.003±0.000	0.003±0.000	0.003±0.000	0.004±0.000	0.004±0.000	0.004±0.000	0.004±0.000
	DC	0.038±0.001	0.093±0.001	0.099±0.000	0.069±0.001	0.038±0.001	0.059±0.001	0.082±0.002	0.052±0.001
	EGuR-3	**0.001±0.000**	**0.001±0.000**	**0.001±0.000**	0.005±0.000	0.001±0.000	0.002±0.000	0.001±0.000	**0.001±0.000**
Qwen3-Next-80B	CodeAct	0.096±0.003	0.096±0.003	0.096±0.003	0.096±0.003	0.083±0.003	0.083±0.003	0.083±0.003	0.083±0.003
	EGuR-ZS	0.073±0.005	0.073±0.005	0.073±0.005	0.073±0.005	0.078±0.009	0.078±0.009	0.078±0.009	**0.078±0.009**
	Mem0	0.071±0.000	0.066±0.000	0.067±0.000	0.063±0.000	0.100±0.000	0.118±0.000	0.102±0.000	0.110±0.000
	EGuR-3	**0.047±0.000**	**0.049±0.000**	**0.043±0.000**	**0.056±0.000**	**0.068±0.000**	**0.052±0.000**	**0.053±0.000**	0.083±0.000

Table 8: Cost ($) across training progress for different methods on Object Counting dataset. Values shown as mean ± standard deviation across seeds. Claude uses EGuR-5, GPT and Qwen use EGuR-3. **Bold** indicates lowest cost within each model, underlined indicates second lowest within each model.

Model	Method	Object Count.			
		0%	33%	66%	100%
Claude-3.5-Sonnet	CodeAct	**0.095±0.005**	0.095±0.005	0.095±0.005	0.095±0.005
	EGuR-ZS	0.234±0.020	0.234±0.020	0.234±0.020	0.234±0.020
	Mem0	0.125±0.001	0.127±0.005	0.125±0.005	0.128±0.009
	DC	1.539±0.046	8.155±0.295	8.803±0.931	7.156±0.397
	EGuR-5	0.242±0.003	**0.060±0.001**	**0.063±0.001**	**0.064±0.001**
GPT-OSS-120B	CodeAct	0.020±0.001	0.020±0.001	0.020±0.001	0.020±0.001
	EGuR-ZS	0.006±0.001	0.006±0.001	**0.006±0.001**	0.006±0.001
	Mem0	0.020±0.003	0.022±0.001	0.021±0.003	0.019±0.000
	DC	0.200±0.009	0.250±0.019	0.194±0.010	0.128±0.048
	EGuR-3	**0.005±0.000**	**0.004±0.000**	0.006±0.000	**0.006±0.001**
Qwen3-Next-80B	CodeAct	0.049±0.001	0.049±0.001	0.049±0.001	0.049±0.001
	EGuR-ZS	0.086±0.010	0.086±0.010	0.086±0.010	0.086±0.010
	Mem0	0.055±0.000	0.057±0.000	0.064±0.000	0.065±0.000
	EGuR-3	**0.049±0.000**	**0.032±0.000**	**0.029±0.000**	**0.025±0.000**

BBEH Object Counting strategy from Claude 3.7 Sonnet before training

```
async def strategy(x):
    SYSTEM_PROMPT = """You are an expert at solving counting and categorization
problems with Python code.

Task: Solve the question by iteratively writing and executing Python code.

Rules:
- Reply with code inside a single ```python ...``` block. The code will be executed
  for you; anything you print will be returned to you next round.
- Read the execution result, and then decide on how to proceed next. You must iterate
  until you have solved the problem.
- When ready, return the solution exactly as: FINAL ANSWER: <answer>
  (You may print this from code or write it directly outside the code block.)

Constraints:
- Code must be self-contained, use only the Python standard library, and avoid
  network/file I/O and pip installs.

Approach:
1. Parse the text to identify all items the narrator ("I") has
2. Categorize each item as either a musical instrument or an animal/insect
3. Count the total number of each category
4. Calculate the absolute difference between the two totals"""

    PARSE_ERROR_PROMPT = "Parsing error: I couldn't find a ```python ... ``` code \
    block or a line starting with 'FINAL ANSWER:' in the model output. \
```

```
    Please either:\n1) Provide valid Python within ```python ... ``` OR\n2) \
    Reply directly with a line exactly: FINAL ANSWER: <answer> with no trailing text."

    max_rounds = 20
    messages = [{"role": "system", "content": SYSTEM_PROMPT}]

    messages.append({"role": "user", "content": f"Problem: {x}"})
    out = await model_call(messages, temperature=0.7, thinking=True, max_tokens=20000)
    messages.append({"role": "assistant", "content": out})
    execution_state = {}
    for i in range(max_rounds):
        # First check for any output code to execute
        blocks = extract_python_code(out)
        if blocks:
            block = "\n\n".join(blocks)
            exec_res, execution_state = python_exec(block, execution_state)

            # Check for final answer in execution output
            m = extract_final_answer(exec_res)
            if m:
                return m.strip()

            # No final answer, so take another turn of model output
            messages.append({
                "role": "user",
                "content": f"The output of executing the generated Python code \
                    is:\n{exec_res}"})
            out = await model_call(
                messages, temperature=0.7, thinking=True, max_tokens=20000)
            messages.append({"role": "assistant", "content": out})
            continue

        # If no code, then check for a final answer
        m = extract_final_answer(out)
        if m:
            return m.strip()
        messages.append({"role": "user", "content": PARSE_ERROR_PROMPT})
        out = await model_call(messages, temperature=0.7, thinking=True, max_tokens=20000)
        messages.append({"role": "assistant", "content": out})

    messages.append({
        "role": "user",
        "content": "Now output the final answer on a single line as \
                   'FINAL ANSWER: answer' with no trailing text."})
    out = await model_call(messages, temperature=0.7, thinking=True, max_tokens=20000)
    m = extract_final_answer(out)
    if m:
        return m.strip()
    else:
        raise Exception("No answer found.")
```

BBEH Object Counting strategy from Claude 3.7 Sonnet after training

```
async def strategy(x):
    PROMPT = """Solve the following problem step by step:

1. First, carefully read through the text and identify all items that "I"
   (the narrator) have. Ignore items that belong to other people
   (grandmother, friend, sister, brother, father, mother, aunt, grandfather,
   cousin, uncle).
2. For each item that belongs to "I", categorize it as either:
   - An animal/insect (e.g., ducks, rabbits, walrus, hens, moose, wasps, bees,
     chameleons, newts, vultures, otter, tigers, shrimps, elks, baboons, yaks,
     boa snakes, anteaters, magpies, fish, narwhals, chimpanzees, antelopes)
   - A musical instrument (e.g., harps, panpipes, balafons, marimbas,
     electric guitars, recorders, clavichord, electric pianos, tanpuras, ouds,
```

```
      bass guitars)
   - Other items (cars like nissan pathfinder, kia niro, honda accord,
     nissan frintier, toyota highlander, hyundai elantra, mazda2, buick enclave,
     hyundai santa fe) which we will ignore for this problem
3. Create a table with columns: Item, Category (Animal/Insect, Musical Instrument,
   Other), Quantity.
4. For items with stories about changing quantities, ALWAYS use the FINAL quantity
   mentioned. For example:
   - "initially I had X, then I lost Y, but then I got Z, but then I lost W,
      so at the end I am left with V" → Count as V
   - "initially I had X, then I also got Y making it a total of Z, then I got W
      making it a total of A, but then I lost B of them so eventually I am left
      with C" → Count as C
5. Count the total number of animals/insects that "I" have.
6. Count the total number of musical instruments that "I" have.
7. Calculate the absolute difference between these two totals.

Be methodical and organize your work clearly. After completing your analysis,
write the final answer on a single line as 'FINAL ANSWER: <answer>' with no trailing
text.

Problem: {x}"""

    MISSING_ANSWER_FEEDBACK = "Your last response did not include a line exactly of \
        the form 'FINAL ANSWER: <answer>'. Please try again to output a final answer,\
        responding with exactly one line 'FINAL ANSWER: <answer>'."

    messages = [{"role": "user", "content": PROMPT.format(x=x)}]
    for _ in range(5):
        output = await model_call(
            messages, temperature=0.7, thinking=True, max_tokens=20000)
        # Find FINAL ANSWER
        ans = extract_final_answer(output)
        if ans:
            return ans
        messages.append({"role": "assistant", "content": output})
        messages.append({"role": "user", "content": MISSING_ANSWER_FEEDBACK})

    # One last attempt with a more direct prompt
    messages.append({
        "role": "user",
        "content": "Based on your analysis, what is the absolute difference between \
        the total number of animals/insects and musical instruments that the narrator \
        has? Provide only the numerical answer in the format 'FINAL ANSWER: <number>'."})
    output = await model_call(
        messages, temperature=0.7, thinking=True, max_tokens=20000)
    ans = extract_final_answer(output)
    if ans:
        return ans
    else:
        raise Exception("No answer found after multiple attempts")
```

E Example Learned Context

The final memory resulting from EGUR-5 with Claude 3.7 Sonnet on BBEH Movie Recommendations is shown below. The memory consists of several memory entries and a single entry containing the (task, best strategy) pair which is used for solving future problems.

Learned Context

```
<memory_entry-1>
When analyzing similarity between items for group appeal (movies, books, etc.), consider these key
  ↪  factors:
1. Era/time period consistency (items from the same era tend to have more similar appeal)
```

```
2. Language/accessibility (foreign language items create significant outliers)
3. Target demographics (mainstream vs. niche appeal)
4. Critical reception and popular appeal
5. Production values and stylistic approaches

For movie similarity specifically, the most reliable indicators of similar group appeal are:
- Same language and era
- Similar accessibility factors
- Comparable mainstream vs. niche positioning
- Consistent tone and pacing
</memory_entry-1>

<memory_entry-2>
For similarity analysis problems, the Chain of Thought (CoT) approach with deep thinking tends to be more cost-effective than more complex strategies like CodeAct or Evaluator-Optimizer when the task involves subjective judgment rather than algorithmic calculation.

The CoT approach with a well-structured prompt that guides the analysis through specific criteria (genres, audience, reception, etc.) and asks for explicit rating/ranking of options provides sufficient structure for the model to make accurate comparisons without requiring the overhead of code execution or multiple model calls.
</memory_entry-2>

[...]

<memory_entry-5>
Task: Analyzing which group of movies has the most similar audience appeal
Best Strategy:
<strategy>
async def strategy(x):
    PROMPT = f"""You are an expert film critic tasked with analyzing which group of movies would have the most similar appeal to the same audience. Think step by step and provide your final answer on a single line as 'FINAL ANSWER: <option letter>' with no trailing text.

When analyzing similarity between movies for group appeal, consider these key factors:
1. Era/time period consistency (movies from the same era tend to have more similar appeal)
2. Language/accessibility (foreign language films create significant outliers)
3. Target demographics (mainstream vs. niche appeal)
4. Critical reception and popular appeal
5. Production values and stylistic approaches
6. Psychological appeal patterns (puzzle-solving vs. emotional engagement vs. escapism)
7. Viewing experience requirements (active vs. passive engagement)
8. Narrative complexity and accessibility
9. Emotional responses elicited (fear, humor, intellectual stimulation, etc.)

For movie similarity specifically, the most reliable indicators of similar group appeal are:
- Same language and era
- Similar accessibility factors
- Comparable mainstream vs. niche positioning
- Consistent tone and pacing

Significant outliers that reduce group cohesion include:
- Animated shorts among feature films
- Family/children's content mixed with adult-oriented films
- Critically panned films grouped with acclaimed ones
- Comedic/light entertainment mixed with serious dramas
- Foreign language films mixed with English-language films
- Adult/erotic content mixed with general audience films

For each option:
1. Analyze each movie in the group individually
2. Compare each movie against every other movie in the group
3. Identify specific outliers that would reduce group cohesion
4. Rate the overall cohesion on a 1-10 scale
5. Explain your reasoning thoroughly

Problem: {x}

```

```
Think through each option carefully and systematically.
"""
    MISSING_ANSWER_FEEDBACK = "Your last response did not include a line exactly of the form 'FINAL ANSWER: <option letter>'. Please try again to output a final answer, responding with exactly one line 'FINAL ANSWER: <option letter>'."

    messages = [{"role": "user", "content": PROMPT}]
    for _ in range(5):
        output = await model_call(messages, temperature=1.0, thinking=True, max_tokens=20000)
        # Find FINAL ANSWER
        ans = extract_final_answer(output)
        if ans:
            return ans
        messages.append({"role": "assistant", "content": output})
        messages.append({"role": "user", "content": MISSING_ANSWER_FEEDBACK})
    raise Exception("No answer found")
</strategy>
</memory_entry-5>
```

F Prompts

We include the complete prompts for the Guide and Consolidator in the following sections. These prompts define the two strategies defining EGUR.

F.1 Guide

The prompt for the Guide is included below.

Guide Prompt

```
You are designing a **reasoning strategy**, *not* solving the target problem.

A reasoning strategy is a **Python function** that describes the design of an LLM workflow/agent to solve a particular problem.
It should **not** directly contain or compute the final answer to the target problem. Instead, it should provide the **workflow/environment** for an LLM system to arrive at the correct answer.
The strategy doesn't need to describe the explicit steps to take to solve the problem and can instead just set up an agent which can figure out how to solve the problem itself (see the CodeAct example below).
Once you output a runnable strategy, you will be given the reasoning trace from the strategy execution and you can optionally update any saved memory.

**Definition:**
A reasoning strategy is an async function `strategy(x)` where:
- `x` (string) is the problem statement.
- The function can contain calls to the following functions:
    - `model_call: (list[dict], temperature, thinking, max_tokens) -> Awaitable[str]`: an async call to a model which takes conversation history (list of dict) and some keyword generation parameters as input and outputs the response. You may use asyncio.gather to perform concurrent calls to the model.
    - `extract_python_code: str -> Optional[List[str]]`: a parsing function to extract all code from Python markdown blocks using the regex r"```(?:python|py)\s*(.*?)\s*```".
    - `extract_final_answer: str -> Optional[str]`: a parsing function to extract the answer for text in the format of "FINAL ANSWER: answer".
    - `python_exec: (str, dict) -> (str, dict)`: a function which takes the code string and state dict as input and returns the output of executing the code as well as the updated state dict. This can only execute python code using standard libraries, so it does not support any imports which must be installed with pip.
- The function can use global variables `MEMORY_ENTRY_K` which are set to the string containing the entire body of <memory_entry-k></memory_entry-k>.
- The strategy function must return the final answer as a string.

**Important:**
```

- **Do not** attempt to answer the problem yourself or write code without LLM calls which solves the problem.
- **Do not** implement the `python_exec` function or the `extract_final_answer` function as it will be provided if you need it.
- Your output must define the `strategy` Python function in a <strategy></strategy> block**.
- The function must be executable without changes and cannot import any non-standard libraries.
- If including code in a prompt, be careful about including quotes within a string. Use different types of quotes for the outer and inner quotes or explicitly escape all the inner quotes.
- The strategy should return the final answer only with no other text.
- You should reference any available memory below to help design the best strategy.

Strategy motifs
Some commonly used elements which can be composed together to create the reasoning strategy:
- chain of thought (by asking to think step by step)
- code interpreter (using python_exec)
- iterative feedback to improve answers or correct any possible runtime errors (feeding the output from the model back to itself in case of error)
- majority voting (sample multiple times with temperature and take the most common answer)
- deep thinking mode (enabled with the thinking=True flag to `model_call` which automatically sets temperature to 1.0. Temperature MUST BE greater than 0 (preferably 0.6-1.0) when thinking=True.)
- problem decomposition (first split to subproblems, then solve each subproblem either with a separate LLM call or all at once)
- evaluator and optimizer (keep trying to solve until the proposed answer is deemed correct by a verification method which can be implemented as a tool call or LLM call)

Example implementations of some strategies:
<example>
CoT with deep thinking (a single prompt to an LLM to solve the problem with retries if answer is missing)
<strategy>
```python
async def strategy(x):
    PROMPT = "Solve the following problem step by step and write the final answer on a single line as 'FINAL ANSWER: <answer>' with no trailing text:\nProblem: {x}"
    MISSING_ANSWER_FEEDBACK = "Your last response did not include a line exactly of the form 'FINAL ANSWER: <answer>'. Please try again to output a final answer, responding with exactly one line 'FINAL ANSWER: <answer>'."

    messages = [{"role": "user", "content": PROMPT.format(x=x)}]
    for _ in range(10):
        output = await model_call(messages, temperature=1.0, thinking=True, max_tokens=20000) # Note that you can set thinking=False to turn off deep thinking for simpler problems
        # Find FINAL ANSWER
        ans = extract_final_answer(output)
        if ans:
            return ans
        messages.append({"role": "assistant", "content": output})
        messages.append({"role": "user", "content": MISSING_ANSWER_FEEDBACK})
    raise Exception("No answer found")
```
</strategy>
</example>

<example>
CodeAct (loop of writing code and executing the code until answer is reached)
<strategy>
```python
async def strategy(x):
    SYSTEM_PROMPT = """You are an expert at solving problems with Python code.

Task: Solve the question by iteratively writing and executing Python code.

Rules
- Reply with code inside a single ```python ...``` block. The code will be executed for you; anything you print will be returned to you next round.
- Read the execution result, and then decide on how to procede next. You must iterate until you have solved the problem.
- When ready, return the solution exactly as: FINAL ANSWER: <answer>
```

```
  (You may print this from code or write it directly outside the code block.)

Constraints
- Code must be self-contained, use only the Python standard library, and avoid network/file I/O and pip installs."""
    PARSE_ERROR_PROMPT = "Parsing error: I couldn't find a ```python ... ``` code block or a line starting with 'FINAL ANSWER:' in the model output. Please either:\n1) Provide valid Python within ```python ... ``` OR\n2) Reply directly with a line exactly: FINAL ANSWER: <answer> with no trailing text."

    max_rounds = 20
    messages = [{"role": "system", "content": SYSTEM_PROMPT}]

    messages.append({"role": "user", "content": f"Problem: {x}"})
    out = await model_call(messages, temperature=1.0, thinking=True, max_tokens=20000)
    messages.append({"role": "assistant", "content": out})
    execution_state = {}
    for i in range(max_rounds):
        # First check for any output code to execute
        # We must always first execute any code or else we may parse out a "FINAL ANSWER" print statement before executing it
        blocks = extract_python_code(out)
        if blocks:
            block = "\n\n".join(blocks)
            exec_res, execution_state = python_exec(block, execution_state)

            # Check for final answer in execution output
            m = extract_final_answer(exec_res)
            if m:
                return m.strip()

            # No final answer, so take another turn of model output
            messages.append({"role": "user", "content": f"The output of executing the generated Python code is:\n{exec_res}"})
            out = await model_call(messages, temperature=1.0, thinking=True, max_tokens=20000)
            messages.append({"role": "assistant", "content": out})
            continue

        # If no code, then check for a final answer
        m = extract_final_answer(out)
        if m:
            return m.strip()
        messages.append({"role": "user", "content": PARSE_ERROR_PROMPT})
        out = await model_call(messages, temperature=1.0, thinking=True, max_tokens=20000)
        messages.append({"role": "assistant", "content": out})

    messages.append({"role": "user", "content": "Now output the final answer on a single line as 'FINAL ANSWER: answer' with no trailing text."})
    out = await model_call(messages, temperature=1.0, thinking=True, max_tokens=20000)
    m = extract_final_answer(out)
    if m:
        return m.strip()
    else:
        raise Exception("No answer found.")
```
</strategy>
</example>

<example>
Parallelization (sample multiple times with temperature and then take the most common response. For very ambiguous or challenging problems, sampling 20+ times can be helpful):
<strategy>
```
async def strategy(x):
    PROMPT = "Solve the problem and output the final answer on a single line as 'FINAL ANSWER: <answer>' with no trailing text.\nProblem: {x}"

    outputs = []
    messages = [{"role": "user", "content": PROMPT.format(x=x)}]
    async def get_response():
```

```
        # use higher temperature for more variation
        out = await model_call(messages, temperature=1.0, thinking=True, max_tokens=20000)

        # Find FINAL ANSWER
        ans = extract_final_answer(out)
        if ans:
            return ans
        else:
            return "No answer found"

    # multiple independent calls can run concurrently with asyncio.gather
    outputs = await asyncio.gather(*[get_response() for _ in range(10)])

    # return the majority of the answers
    return max(set(outputs), key = outputs.count)
</strategy>
</example>

<example>
Evaluator-Optimizer (one LLM call generates a solution and another provides evaluation and feedback for the generator to improve its solution)
<strategy>
```python
async def strategy(x):
    GENERATOR_PROMPT = "You are an expert problem solver. Solve the given problem step by step and provide your final answer on a single line as 'FINAL ANSWER: <answer>' with no trailing text."
    EVALUATOR_PROMPT = "Evaluate whether the following answer to a problem is correct.\nProblem: {problem}\nProposed Answer: {solution}\nIf the answer is correct, respond with exactly 'FINAL ANSWER: CORRECT'. If the answer is incorrect, respond with 'FINAL ANSWER: <explanation of what is wrong and guidance for improvement>'."
    MISSING_ANSWER_PROMPT = "The last output is missing a line of the form 'FINAL ANSWER: ...' which needs to be parsed out. Please try again to output a line following this format."

    max_iterations = 10

    # Generate initial solution
    generator_messages = [
        {"role": "system", "content": GENERATOR_PROMPT},
        {"role": "user", "content": f"Problem: {x}"}
    ]

    for i in range(max_iterations):
        # get answer from generator
        for _ in range(5):
            solution = await model_call(generator_messages, temperature=1.0, thinking=True, max_tokens=20000)
            generator_messages.append({"role": "assistant", "content": solution})

            # Check if we have a final answer
            ans = extract_final_answer(solution)
            if ans:
                break
            else:
                generator_messages.append({"role": "user", "content": MISSING_ANSWER_PROMPT})

        # get evaluation of the current answer
        eval_messages = [{"role": "user", "content": EVALUATOR_PROMPT.format(problem=x, solution=ans)}]
        for _ in range(5):
            evaluation = await model_call(eval_messages, temperature=1.0, thinking=True, max_tokens=20000)

            eval_ans = extract_final_answer(evaluation)
            if eval_ans:
                break
            else:
                eval_messages = [{"role": "assistant", "content": evaluation}]
                eval_messages.append({"role": "user", "content": MISSING_ANSWER_PROMPT})

        # only return the answer if the evaluator says it is correct
```
```

```
        if "CORRECT" in eval_ans:
            return ans

        # otherwise, make the generator improve the answer based on the evaluator output
        generator_messages.append({"role": "user", "content": f"An evaluator gave the following feedback on your answer: {evaluation}. Please revise your answer and output the new answer on a line as 'FINAL ANSWER: answer'."})

    # Final attempt to extract answer
    solution = await model_call(generator_messages, temperature=1.0, thinking=True, max_tokens=20000)
    ans = extract_final_answer(solution)
    if ans:
        return ans.strip()
    else:
        raise Exception("No answer found.")
```
</strategy>
</example>

If parsing anything from the output of the model, be sure to exactly specify the format the model is supposed to respond in and regenerate using any failed parsing feedback if there are parsing errors.
It's best to keep the max_tokens around 20000 for all calls to `model_call` to avoid issues from output truncation, unless the memory shows that using lower max_tokens is safe.
Your central goal is to write a strategy that will correctly solve the problem, but if there are two possible strategies which are equally likely to be correct, favor the less costly strategy.
A strategy should never give up! The strategy must solve the provided problem even if it takes a long time, so the strategy should take whatever steps necessary to solve the problem correctly and be sure it does not result in giving up and outputting an incorrect answer. If everything goes wrong, then throw an exception.
Strategies are written at the meta-level (see how the above examples are not problem specific), and you should only tailor prompts to the problem if you are sure it will help.
Note that for many non-algorithmic tasks, a simple CoT approach (with in-context examples if available) is one of the best strategies provided the model is capable enough, and CodeAct is one of the best strategies for algorithmic tasks.

Memory
[[PAST]]
Use the above memory to help design the strategy. It is your job to incorporate any useful information from the memory into the strategy. This can be incorporated into the strategy at two levels:
1. The high level structure (the memory provides the best strategies for related tasks as well as other notes about useful strategies). Base the strategy structure on the memory as much as possible.
2. The prompts (e.g. include all important advice, code blocks to use, pitfalls to avoid, few-shot examples, etc. which the model should have in its context). To include a memory entry into a strategy (such as to include code into a prompt), use the global variable `MEMORY_ENTRY_K` where K is replaced with the id of the entry or copy the content of the memory verbatim. Include as much information from the notes as possible into prompts to help the models avoid pitfalls and streamline their reasoning.
Please mention if you are using any of the above memories when designing the strategy. Before designing a new strategy, check if a (task, best strategy) pair exists in the memory for the current task and use the best strategy if so.
Ignore irrelevant memory entries.
Overall, you want to think about how to use more computation to avoid incorrect answers when they occur (more verification/CodeAct iterations/voting/thinking when the problem is very hard or past problems were answered incorrectly) and when to use less computation (simpler strategies such as CoT when the problem/subproblem is simple or past problems were solved easily) to reduce cost.
You want to produce the strategy most likely to get a correct answer.

Target problem:
[[PROBLEM]]

Your Task:
[[TASK_PROMPT]]

F.2 Consolidator

The prompt for the Consolidator is included below.

Consolidator Prompt

```
The outcomes from executing the generated strategies are shown below.
<experience>
<strategy>
    [[PAST_STRATEGY]]
</strategy>
<trace>
    [[STRATEGY_EXECUTION_TRACE]]
</trace>
<final_answer>[FINAL_ANSWER]</final_answer>
<verifier_output>[[Answer is correct! or Answer is incorrect!]]</verifier_output>
<cost>[[COST]]</cost>
</experience>
...

Your task is now to update a memory to help you solve problems better in the future by generating better strategies.
The memory consists of (task, best strategy) pairs where there is exactly one entry per task seen as well as entries about general useful findings to improve strategies in the future.

**Workflow**
1. For each experience, examine the verifier output and find the strategies that resulted in the correct answer. Of these correct answer strategies, find the one which has greatest reliability or lowest cost. If all strategies resulted in the wrong answer, then continue to step 2 and DO NOT SAVE THE STRATEGY to memory.
    1.1. Check the current memory for existing (task, best strategy) pairs where the task matches the current problem.
    1.2. If there is an existing (task, best strategy) entry in the memory for this task, update the memory entry with the strategy found in step 1 ONLY IF the strategy used above is better (in reliability or lower cost) than the one already in memory.
2. From the reasoning traces of the strategies extract any useful information (such as code snippets and useful examples) which could help in the future. DO NOT SAVE any strategy which resulted in the incorrect answer.

**Extract**
- The best strategies that resulted in correct answers, helpful functions, heuristics, examples. Feel free to save code.
- Unexpected behaviors of models or tools in the reasoning trace which should be remembered to help design better strategies and avoid unneccessary rework or continued failures in the future
- When certain strategy motifs are useful or harmful
- Useful and working code snippets, tool configurations which can be provided in the future rather than rediscovered
- Anything that would save significant reasoning time/cost (such as having to reimplement a code solution or reducing the number of reasoning steps)
- Failure modes and their early detection/mitigation
- Effective prompts or verification methods
- Cost-saving shortcuts that maintain accuracy
- Failures in tool-use which could be fixed in the strategy

**Avoid**
- Problem-specific literals (numbers, names, dataset IDs) or any information which would not apply to other problems
- Obvious insights, redundant info, incomplete code
- Anything that requires rework, for instance if something was derived in the reasoning trace (like a code solution) then don't say "solve with code" and instead save the actual code.
- Inventing or coming up with anything new, for instance if the reasoning trace shows a failed solution, do not fix the solution now since your new solution may also be wrong. Instead, it is better to examine the reasoning trace to determine why it was wrong or what was an unsuccessful strategy to avoid the mistake in the future.
- Saving strategies that resulted in the wrong answer

**Output**
To add or update the (task, best strategy) pair, output blocks of the form:
<memory_entry>
Task: [short description of the task to allow you to know which problems to use this strategy for in the future. Can be at any granularity that is helpful.]
Best Strategy:
```

```
<strategy>
[Code for the best strategy discovered for the task]
</strategy>
</memory_entry>
For saving other useful information, but not full strategies, output blocks of the form:
<memory_entry>
[Actionable memory entry to improve future behavior on similar problems.]
</memory_entry>

To delete memory entries, output blocks of the form:
<delete_entry>
[mem-id]
</delete_entry>
For instance:
<delete_entry>
1
</delete_entry>
will delete the memory entry block of the form <memory_entry-1></memory_entry-1>.
When you output a new memory, DO NOT add the ID to the memory_entry block as the ID will be automatically assigned for you.
To edit a memory, delete the old memory and then output the updated memory.
Never delete a (task, best strategy) pair from memory.
Be careful about deleting entries because other memory entries may refer to the deleted memory entry or even use the variable `MEMORY_ENTRY_K` within code.

**Existing Memory**
[[past_memories]]
Memory length: [[memory_len]] tokens

Output nothing if you do not want to change anything.
Try not to remove any memories which could be useful in the future.
Also, ensure the memory stays shorter than 10000 tokens or else it will distract more than it helps. If the memory is above 10k tokens, try to only edit the memory rather than make additions.
Before outputting the memory edits, write a short justification for your changes.
```