AGENTIC CONTEXT ENGINEERING: LEARNING COMPREHENSIVE CONTEXTS FOR SELF-IMPROVING LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) applications such as agents and domain-specific reasoning increasingly rely on context adaptation, modifying model inputs with instructions, strategies, or evidence, rather than weight updates. While prior methods improve usability, they often suffer from a brevity bias, discarding domain-specific insights in favor of short summaries, and from context collapse, where iterative rewriting erodes details over time. Building on the adaptive memory introduced by Dynamic Cheatsheet, we present ACE (**A**gentic **C**ontext **E**ngineering), a framework that treats contexts as evolving playbooks that accumulate, refine, and organize strategies through a modular process of generation, reflection, and curation. ACE prevents collapse by applying structured, incremental updates that preserve detailed knowledge and scale with long-context models. Across agentic and domain-specific benchmarks, ACE consistently outperforms strong baselines, improving application performance by 9.0% while reducing adaptation latency and rollout cost. Notably, ACE could adapt effectively without labeled supervision, instead leveraging natural execution feedback, and on the AppWorld leaderboard it matches the top-1-ranked production-level agent while using a smaller open-source model. These results demonstrate that comprehensive, evolving contexts enable scalable, efficient, and self-improving LLM systems.

1 INTRODUCTION

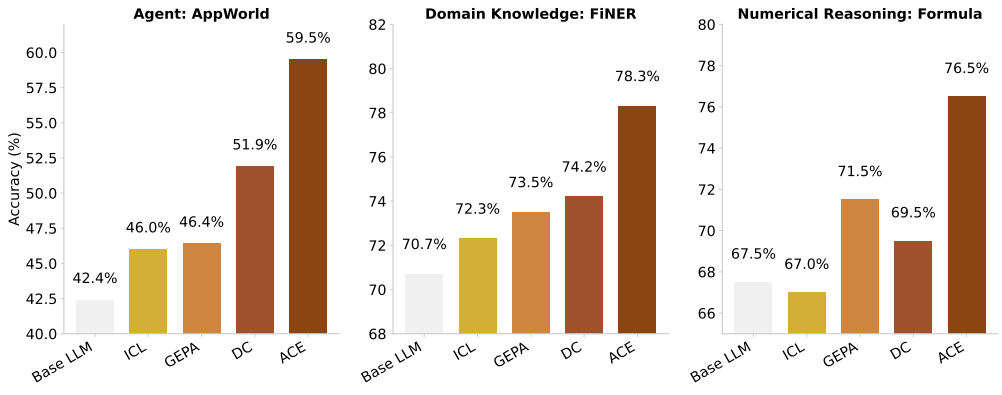

Figure 1: **Overall Performance Results.** Our proposed framework, ACE, consistently outperforms strong baselines across agent and domain-specific tasks.

Modern AI applications based on large language models (LLMs), such as LLM agents (Yao et al., 2023; Yang et al., 2024) and compound AI systems (Zaharia et al., 2024), increasingly depend on *context adaptation*. Instead of modifying model weights, context adaptation improves performance after model training by incorporating clarified instructions, structured reasoning steps, or domain-specific input formats directly into the model's inputs. Contexts underpin many AI system components, including system prompts that guide downstream tasks (Opsahl-Ong et al., 2024; Agrawal

et al., 2025), memory that carries past facts and experiences (Suzgun et al., 2025b; Xu et al., 2025), and factual evidence that reduces hallucination and supplements knowledge (Asai et al., 2024).

Adapting through *contexts* rather than *weights* offers several key advantages. Contexts are interpretable and explainable for users and developers (Wei et al., 2022; Wang et al., 2022), allow rapid integration of new knowledge at runtime (Lewis et al., 2020; Borgeaud et al., 2022), and can be shared across models or modules in a compound system (Khot et al., 2022). Meanwhile, advances in long-context LLMs (Peng et al., 2023) and context-efficient inference such as KV cache reuse (Gim et al., 2024; Yao et al., 2025) are making context-based approaches increasingly practical for deployment. As a result, context adaptation is emerging as a central paradigm for building capable, scalable, and self-improving AI systems.

Despite this progress, existing approaches to context adaptation face two key limitations. First, a *brevity bias*: many prompt optimizers prioritize concise, broadly applicable instructions over comprehensive accumulation. For example, GEPA (Agrawal et al., 2025) highlights brevity as a strength, but such abstraction can omit domain-specific heuristics, tool-use guidelines, or common failure modes that matter in practice (Gao et al., 2025). This objective aligns with validation metrics in some settings, but often fails to capture the detailed strategies required by agents and knowledge-intensive applications. Second, *context collapse*: methods that rely on monolithic rewriting by an LLM often degrade into shorter, less informative summaries over time, causing sharp performance declines (Figure 2). In domains such as interactive agents (Trivedi et al., 2024; Patil et al., 2024; Zhang et al., 2024), domain-specific programming (Ye et al., 2023; Zhang et al., 2025a), and financial or legal analysis (Loukas et al., 2022; Guha et al., 2023; Wang et al., 2025), strong performance depends on retaining detailed, task-specific knowledge rather than compressing it away.

As applications such as agents and knowledge-intensive reasoning demand greater reliability, recent work has shifted toward saturating contexts with abundant, potentially useful information (Jiang et al., 2025; Chung et al., 2025; Chen et al., 2025), enabled by advances in long-context LLMs (Peng et al., 2023; Mao et al., 2024). **We argue that contexts should function not as concise summaries, but as comprehensive, structured playbooks—detailed, inclusive, and rich with domain insights.** Unlike humans, who often benefit from concise generalization, LLMs are more effective when provided with long, detailed contexts and can distill relevance autonomously (Jiang et al., 2025; Liu et al., 2025; Suzgun et al., 2025b). Thus, instead of compressing away domain-specific heuristics and tactics, contexts should preserve them, allowing the model to decide what matters at inference time.

To address these limitations, we introduce ACE (**A**gentic **C**ontext **E**ngineering), a framework for comprehensive context adaptation in both offline settings (*e.g.,* system prompt optimization) and online settings (*e.g.,* test-time memory adaptation). Rather than compressing contexts into distilled summaries, ACE treats them as evolving playbooks that accumulate and organize strategies over time. Building on the agentic architecture of Dynamic Cheatsheet (Suzgun et al., 2025b), ACE incorporates a modular workflow of generation, reflection, and curation, while adding structured, incremental updates guided by a grow-and-refine principle. This design preserves detailed, domain-specific knowledge, prevents context collapse, and yields contexts that remain comprehensive and scalable throughout adaptation.

We evaluate ACE on two categories of LLM applications that most benefit from comprehensive, evolving contexts: (1) *agents* (Trivedi et al., 2024), which require multi-turn reasoning, tool use, and environment interaction, where accumulated strategies can be reused across episodes; and (2) *domain-specific benchmarks*, which demand specialized tactics and knowledge, where we focus on financial analysis (Loukas et al., 2022; Wang et al., 2025). Our key findings are:

- ACE consistently outperforms strong baselines, yielding average gains of 10.6% on *agents* and 8.6% on *domain-specific benchmarks*, across both offline and online adaptation settings.
- ACE is able to construct effective contexts *without* labeled supervision, instead leveraging execution feedback and environment signals—key ingredients for self-improving LLMs and agents.
- On the AppWorld benchmark leaderboard (AppWorld, 2025), ACE surpasses the top-1-ranked production-level agent IBM-CUGA (Marreed et al., 2025) (powered by GPT-4.1) while using a much smaller open-source model (DeepSeek-V3.1).

- ACE requires significantly fewer rollouts and achieves lower adaptation latency than existing adaptive methods, demonstrating that scalable self-improvement can be achieved with both higher accuracy and lower cost.

2 Background and Motivation

2.1 Context Adaptation

Context adaptation (or context engineering) refers to methods that improve model behavior by constructing or modifying inputs to an LLM, rather than altering its weights. The current state of the art leverages *natural language feedback* (Shinn et al., 2023; Yuksekgonul et al., 2024; Agrawal et al., 2025). In this paradigm, a language model inspects the current context along with signals such as execution traces, reasoning steps, or validation results, and generates natural language feedback on how the context should be revised. This feedback is then incorporated into the context, enabling iterative adaptation. Representative methods include Reflexion (Shinn et al., 2023), which reflects on failures to improve agent planning; TextGrad (Yuksekgonul et al., 2024), which optimizes prompts via gradient-like textual feedback; GEPA (Agrawal et al., 2025), which refines prompts iteratively based on execution traces and achieves strong performance, even surpassing reinforcement learning approaches in some settings; and Dynamic Cheatsheet (Krause et al., 2019), which constructs an external memory that accumulates strategies and lessons from past successes and failures during inference. These natural language feedback methods represent a major advance, offering flexible and interpretable signals for improving LLM systems beyond weight updates.

2.2 Limitations of Existing Context Adaptation Methods

The Brevity Bias. A recurring limitation of context adaptation methods is *brevity bias*: the tendency of optimization to collapse toward short, generic prompts. Gao et al. (Gao et al., 2025) document this effect in prompt optimization for test generation, where iterative methods repeatedly produced near-identical instructions (*e.g.,*, "Create unit tests to ensure methods behave as expected"), sacrificing diversity and omitting domain-specific detail. This convergence not only narrows the search space but also propagates recurring errors across iterations, since optimized prompts often inherit the same faults as their seeds. More broadly, such bias undermines performance in domains that demand detailed, context-rich guidance—such as multi-step agents, program synthesis, or knowledge-intensive reasoning—where success hinges on accumulating rather than compressing task-specific insights.

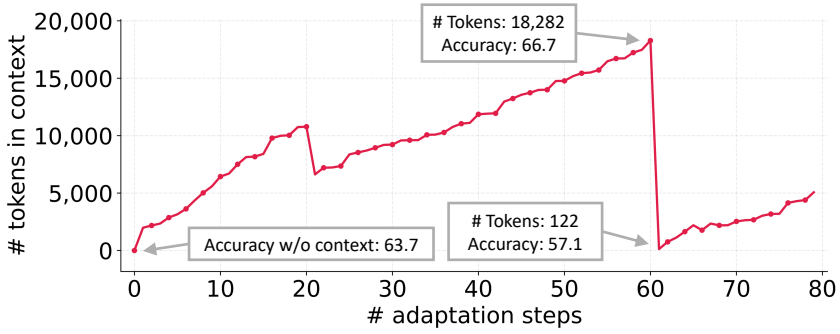

Figure 2: **Context Collapse.** Monolithic rewriting of context by an LLM can collapse it into shorter, less informative summaries, leading to sharp performance drops.

Context Collapse. In a case study on the AppWorld benchmark (Trivedi et al., 2024), we observe a phenomenon we call *context collapse*, which arises when an LLM is tasked with fully rewriting the accumulated context at each adaptation step. As the context grows large, the model tends to compress it into much shorter, less informative summaries, causing a dramatic loss of information. For instance, at step 60 the context contained 18,282 tokens and achieved an accuracy of 66.7, but at the very next step it collapsed to just 122 tokens, with accuracy dropping to 57.1—worse than

the baseline accuracy of 63.7 without adaptation. While we highlight this through Dynamic Cheatsheet (Suzgun et al., 2025b), the issue is not specific to that method; rather, it reflects a fundamental risk of end-to-end context rewriting with LLMs, where accumulated knowledge can be abruptly erased instead of preserved.

The Generated ACE Playbook on AppWorld

STRATEGIES AND HARD RULES

[shr-00009]

When processing time-sensitive transactions involving specific relationships: always resolve identities from the correct source app (phone contacts), use proper datetime range comparisons instead of string matching, and verify all filtering criteria (relationship + time) are met before processing items. This ensures accurate identification and processing of the right transactions.

USEFUL CODE SNIPPETS AND TEMPLATES

[code-00013]

For efficient artist aggregation when processing songs, use defaultdict(list) to map song titles to artist names:

```
from collections import defaultdict artist_map = defaultdict(list) for song in songs:
artist_map[song['title']].extend([artist['name'] for artist in song['artists']])
```

TROUBLESHOOTING AND PITFALLS

[ts-00003]

If authentication fails, troubleshoot systematically: try phone number instead of email as username, clean credentials from supervisor, check API documentation for correct parameters etc. Do not proceed with workarounds.

Figure 3: **Example ACE-Generated Context on the AppWorld Benchmark** (partially shown). ACE-generated contexts contain detailed, domain-specific insights along with tools and code that are readily usable, serving as a comprehensive playbook for LLM applications.

3 Agentic Context Engineering (ACE)

We present ACE (**A**gentic **C**ontext **E**ngineering), a framework for scalable and efficient context adaptation in both offline (*e.g.,* system prompt optimization) and online (*e.g.,* test-time memory adaptation) scenarios. Instead of condensing knowledge into terse summaries or static instructions, ACE treats contexts as evolving playbooks that continuously accumulate, refine, and organize strategies over time. Building on the agentic design of Dynamic Cheatsheet (Suzgun et al., 2025b), ACE introduces a structured division of labor across three roles (Figure 4): the *Generator*, which produces reasoning trajectories; the *Reflector*, which distills concrete insights from successes and errors; and the *Curator*, which integrates these insights into structured context updates. This mirrors how humans learn—experimenting, reflecting, and consolidating—while avoiding the bottleneck of overloading a single model with all responsibilities.

To address the limitations of prior methods discussed in §2.2—notably *brevity bias* and *context collapse*—ACE introduces three key innovations: (1) a dedicated *Reflector* that separates evaluation and insight extraction from curation, improving context quality and downstream performance (§4.5); (2) incremental *delta updates* (§3.1) that replace costly monolithic rewrites with localized edits, reducing both latency and compute cost (§4.6); and (3) a *grow-and-refine* mechanism (§3.2) that balances steady context expansion with redundancy control.

As shown in Figure 4, the workflow begins with the Generator producing reasoning trajectories for new queries, which surface both effective strategies and recurring pitfalls. The Reflector critiques these traces to extract lessons, optionally refining them across multiple iterations. The Curator then synthesizes these lessons into compact *delta entries*, which are merged deterministically into the existing context by lightweight, non-LLM logic. Because updates are itemized and localized,

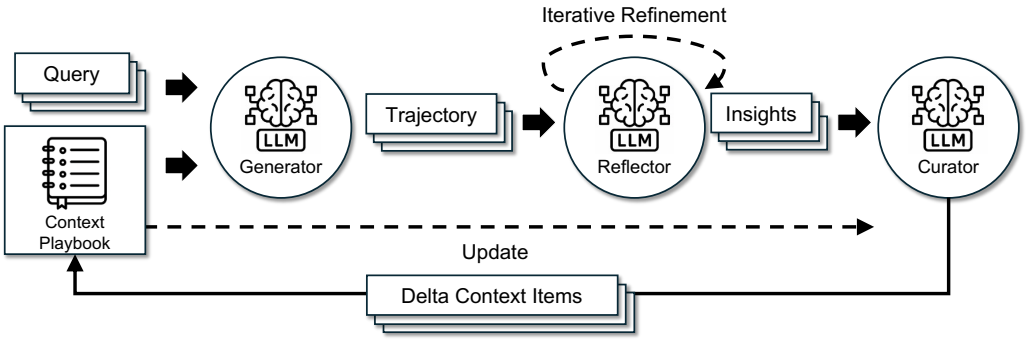

Figure 4: **The ACE Framework.** Inspired by Dynamic Cheatsheet, ACE adopts an agentic architecture with three specialized components: a Generator, a Reflector, and a Curator.

multiple deltas can be merged in parallel, enabling batched adaptation at scale. ACE further supports multi-epoch adaptation, where the same queries are revisited to progressively strengthen the context.

3.1 Incremental Delta Updates

A core design principle of ACE is to represent context as a collection of *structured, itemized bullets*, rather than a single monolithic prompt. The concept of a bullet is similar to the concept of a memory entry in LLM memory frameworks like Dynamic Cheatsheet (Suzgun et al., 2025b) and A-MEM (Xu et al., 2025), but builds on top of that and consists of (1) **metadata**, including a unique identifier and counters tracking how often it was marked helpful or harmful; and (2) **content**, capturing a small unit such as a reusable strategy, domain concept, or common failure mode. When solving new problems, the Generator highlights which bullets were useful or misleading, providing feedback that guides the Reflector in proposing corrective updates.

This itemized design enables three key properties: (1) *localization*, so only the relevant bullets are updated; (2) *fine-grained retrieval*, so the Generator can focus on the most pertinent knowledge; and (3) *incremental adaptation*, allowing efficient merging, pruning, and de-duplication during inference.

Rather than regenerating contexts in full, ACE incrementally produces compact *delta contexts*: small sets of candidate bullets distilled by the Reflector and integrated by the Curator. This avoids the computational cost and latency of full rewrites, while ensuring that past knowledge is preserved and new insights are steadily appended. As contexts grow, this approach provides the scalability needed for long-horizon or domain-intensive applications.

3.2 Grow-and-Refine

Beyond incremental growth, ACE ensures that contexts remain compact and relevant through periodic or lazy refinement. In grow-and-refine, bullets with new identifiers are appended, while existing bullets are updated in place (*e.g.,* incrementing counters). A de-duplication step then prunes redundancy by comparing bullets via semantic embeddings. This refinement can be performed proactively (after each delta) or lazily (only when the context window is exceeded), depending on application requirements for latency and accuracy.

Together, incremental updates and grow-and-refine maintain contexts that expand adaptively, remain interpretable, and avoid the potential variance introduced by monolithic context rewriting.

4 Results

4.1 Tasks and Datasets

We evaluate ACE on two categories of LLM applications that benefit most from a comprehensive and evolving context: (1) *agent benchmarks*, which require multi-turn reasoning, tool use, and en-

vironment interaction, where agents can accumulate and reuse strategies across episodes and environments; and (2) *domain-specific benchmarks*, which demand mastery of specialized concepts and tactics, where we focus on financial analysis as a case study.

- **LLM Agent: AppWorld (Trivedi et al., 2024)** is a suite of autonomous agent tasks involving API understanding, code generation, and environment interaction. It provides a realistic execution environment with common applications and APIs (*e.g.,* email, file system) and tasks of two difficulty levels (normal and challenge). A public leaderboard (AppWorld, 2025) tracks performance, where, at the time of submission, the best system achieved only 60.3% average accuracy, highlighting the benchmark's difficulty and realism.
- **Financial Analysis: FiNER (Loukas et al., 2022) and Formula (Wang et al., 2025)** test LLMs on financial reasoning tasks that rely on the eXtensible Business Reporting Language (XBRL). *FiNER* requires labeling tokens in XBRL financial documents with one of 139 fine-grained entity types, a key step for financial information extraction in regulated domains. *Formula* focuses on extracting values from structured XBRL filings and performing computations to answer financial queries, *i.e.,* numerical reasoning.

Evaluation Metrics. For AppWorld, we follow the official benchmark protocol and report *Task Goal Completion* (TGC) and *Scenario Goal Completion* (SGC) on both the test-normal and test-challenge splits. For FiNER and Formula, we follow the original setup and report accuracy, measured as the proportion of predicted answers that exactly match the ground truth.

All datasets follow the original train/validation/test splits. For *offline* context adaptation, methods are optimized on the training split and evaluated on the test split with pass@1 accuracy. For *online* context adaptation, methods are evaluated sequentially on the test split: for each sample, the model first predicts with the current context, then updates its context based on that sample. The same shuffled test split is used across all methods.

4.2 Baselines and Methods

Base LLM. The base model is evaluated directly on each benchmark without any context engineering, using the default prompts provided by dataset authors. For AppWorld, we follow the official `ReAct` (Yao et al., 2023) implementation released by the benchmark authors, and build all other baselines and methods on top of this framework.

In-Context Learning (ICL) (Agarwal et al., 2024). ICL provides the model with task demonstrations in the input prompt (few-shot or many-shot). This allows the model to infer the task format and desired output without weight updates. We supply all training samples when they fit within the model's context window; otherwise, we fill the window with as many demonstrations as possible.

MIPROv2 (Opsahl-Ong et al., 2024). MIPROv2 is a popular prompt optimizer for LLM applications that works by jointly optimizing system instructions and in-context demonstrations via bayesian optimization. We use the official DSPy implementation (DSPy, 2025b), setting `auto="heavy"` to maximize optimization performance.

GEPA (Agrawal et al., 2025). GEPA (Genetic-Pareto) is a sample-efficient prompt optimizer based on reflective prompt evolution. It collects execution traces (reasoning, tool calls, intermediate outputs) and applies natural-language reflection to diagnose errors, assign credit, and propose prompt updates. A genetic Pareto search maintains a frontier of high-performing prompts, mitigating local optima. Empirically, GEPA outperforms reinforcement learning methods such as GRPO and prompt optimizers like MIPROv2, achieving up to 10–20% higher accuracy with as much as 35× fewer rollouts. We use the official DSPy implementation (DSPy, 2025a), setting `auto="heavy"` to maximize optimization performance.

Dynamic Cheatsheet (DC) (Suzgun et al., 2025b). DC is a test-time learning approach that introduces an adaptive external memory of reusable strategies and code snippets. By continuously updating this memory with newly encountered inputs and outputs, DC enables models to accumulate knowledge and reuse it across tasks, often leading to substantial improvements over static prompting methods. A key advantage of DC is that it does not require ground-truth labels: the model can curate

Method	GT Labels	Test-Normal		Test-Challenge		Average
		TGC↑	SGC↑	TGC↑	SGC↑	
DeepSeek-V3.1 as Base LLM						
ReAct		63.7	42.9	41.5	21.6	42.4
Offline Adaptation						
ReAct + ICL	✓	64.3$_{+0.6}$	46.4$_{+3.5}$	46.0$_{+4.5}$	27.3$_{+5.7}$	46.0$_{+3.6}$
ReAct + GEPA	✓	64.9$_{+1.2}$	44.6$_{+1.7}$	46.0$_{+4.5}$	30.2$_{+8.6}$	46.4$_{+4.0}$
ReAct + ACE	✓	**76.2**$_{+12.5}$	**64.3**$_{+21.4}$	**57.3**$_{+15.8}$	**39.6**$_{+18.0}$	**59.4**$_{+17.0}$
ReAct + ACE	✗	75.0$_{+11.3}$	**64.3**$_{+21.4}$	54.4$_{+12.9}$	35.2$_{+13.6}$	57.2$_{+14.8}$
Online Adaptation						
ReAct + DC (CU)	✗	65.5$_{+1.8}$	**58.9**$_{+16.0}$	52.3$_{+10.8}$	30.8$_{+9.2}$	51.9$_{+9.5}$
ReAct + ACE	✗	**69.6**$_{+5.9}$	53.6$_{+10.7}$	**66.0**$_{+24.5}$	**27.3**$_{+5.7}$	**59.5**$_{+17.1}$

Table 1: **Results on the AppWorld Agent Benchmark.** "GT labels" indicates whether ground-truth labels are available to the Reflector during adaptation. We evaluate the ACE framework against multiple baselines on top of the official ReAct implementation, both for offline and online context adaptation. ReAct + ACE outperforms selected baselines by an average of 10.6%, and could achieve good performance even without access to GT labels.

its own memory from its generations, making the method highly flexible and broadly applicable. We use the official implementation released by the authors (Suzgun et al., 2025a) and set it to use the cumulative mode (DC-CU).

ACE (ours). ACE optimizes LLM contexts for both offline and online adaptation through an agentic context engineering framework. To ensure fairness, we use the same LLM for the Generator, Reflector, and Curator (non-thinking mode of DeepSeek-V3.1 (DeepSeek-AI, 2024)), preventing knowledge transfer from a stronger Reflector or Curator to a weaker Generator. This isolates the benefit of context construction itself. We adopt a batch size of 1 (constructing a delta context from each sample). We set the maximum number of Reflector refinement rounds and the maximum number of epoch in offline adaptation to 5.

4.3 Results on Agent Benchmark

Analysis. As shown in Table 1, ACE consistently improves over strong baselines on the AppWorld benchmark. In the offline setting, ReAct + ACE outperforms both ReAct + ICL and ReAct + GEPA by significant margins (12.3% and 11.9%, respectively), demonstrating that structured, evolving, and detailed contexts enable more effective agent learning than fixed demonstrations or single optimized instruction prompts. These gains extend to the online setting, where ACE continues to outperform prior adaptive methods such as Dynamic Cheatsheet by an average of 7.6%.

In the agent use case, ACE remains effective even *without* access to ground-truth labels during adaptation: ReAct + ACE achieves an average improvement of 14.8% over the ReAct baseline in this setting. This robustness arises because ACE leverages signals naturally available during execution (*e.g.,* code execution success or failure) to guide the Reflector and Curator in forming structured lessons of successes and failures. Together, these results establish ACE as a strong and versatile framework for building self-improving agents that adapt reliably both with and without labeled supervision.

Notably, on the latest AppWorld leaderboard, ReAct + ACE (59.4% average) matches the top-1-ranked IBM CUGA (60.3%), a production-level GPT-4.1–based agent (Marreed et al., 2025), despite using the much smaller open-source model DeepSeek-V3.1. With online adaptation, ReAct + ACE even surpasses IBM CUGA by 8.4% in TGC and 0.7% in SGC on test-challenge, underscoring the effectiveness of ACE in building comprehensive and self-evolving contexts for agents.

Method	GT Labels	FINER (Acc↑)	Formula (Acc↑)	Average
DeepSeek-V3.1 as Base LLM				
Base LLM		70.7	67.5	69.1
Offline Adaptation				
ICL	✓	72.3$_{+1.6}$	67.0$_{-0.5}$	69.6$_{+0.5}$
MIPROv2	✓	72.4$_{+1.7}$	69.5$_{+2.0}$	70.9$_{+1.8}$
GEPA	✓	73.5$_{+2.8}$	71.5$_{+4.0}$	72.5$_{+3.4}$
ACE	✓	**78.3**$_{+7.6}$	**85.5**$_{+18.0}$	**81.9**$_{+12.8}$
ACE	✗	71.1$_{+0.4}$	83.0$_{+15.5}$	77.1$_{+8.0}$
Online Adaptation				
DC (CU)	✓	74.2$_{+3.5}$	69.5$_{+2.0}$	71.8$_{+2.7}$
DC (CU)	✗	68.3$_{-2.4}$	62.5$_{-5.0}$	65.4$_{-3.7}$
ACE	✓	**76.7**$_{+6.0}$	76.5$_{+9.0}$	**76.6**$_{+7.5}$
ACE	✗	67.3$_{-3.4}$	**78.5**$_{+11.0}$	72.9$_{+3.8}$

Table 2: **Results on Financial Analysis Benchmark.** "GT labels" indicates whether ground-truth labels are available to the Reflector during adaptation. With GT labels, ACE achieves consistent improvements in both offline and online settings, highlighting the advantage of structured and evolving contexts for domain-specific reasoning and code generation. However, we also observe that in the absence of reliable feedback signals (*e.g.,* ground-truth labels or execution outcomes), both ACE and other adaptive methods such as Dynamic Cheatsheet may degrade, suggesting that context adaptation depends critically on feedback quality.

4.4 Results on Domain-Specific Benchmark

Analysis. As shown in Table 2, ACE delivers strong improvements on financial analysis benchmarks. In the offline setting, when provided with ground-truth answers from the training split, ACE surpasses ICL, MIPROv2, and GEPA by clear margins (an average of 10.9%), showing that structured and evolving contexts are particularly effective when tasks require precise domain knowledge (*e.g.,* financial concepts, XBRL rules) that goes beyond fixed demonstrations or monolithic optimized prompts. In the online setting, ACE continues to exceed prior adaptive methods such as DC by an average of 6.2%, further confirming the benefit of agentic context engineering for accumulating reusable insights across specialized domains.

Moreover, we also observe that when ground-truth supervision or reliable execution signals are absent, both ACE and DC may degrade in performance. In such cases, the constructed context can be polluted by spurious or misleading signals, highlighting a potential limitation of inference-time adaptation without reliable feedback. This suggests that while ACE is robust under rich feedback (*e.g.,* code execution results or formula correctness in agent tasks), its effectiveness depends on the availability of signals that allow the Reflector and Curator to make sound judgments. We return to this limitation in §5.

4.5 Ablation Study

Table 3 reports ablation studies on the AppWorld benchmark, analyzing how individual design choices of ACE contribute to effective context adaptation. We examine three factors: (1) *the Reflector with iterative refinement*, our addition to the agentic framework beyond Dynamic Cheatsheet, (2) *multi-epoch adaptation*, which refines contexts over training samples multiple times, and (3) *offline warmup*, which initializes the context through offline adaptation before online adaptation begins.

4.6 Cost and Speed Analysis

Due to its support for incremental, "delta" context updates and non-LLM-based context merging and de-duplication, ACE demonstrates particular advantages in reducing the cost (in terms of the number of rollouts or the amount of dollar cost for token ingestion/generation) and latency of adaptation.

Method	GT Labels	Test-Normal		Test-Challenge		Average
		TGC↑	SGC↑	TGC↑	SGC↑	
DeepSeek-V3.1 as Base LLM						
ReAct		63.7	42.9	41.5	21.6	42.4
Offline Adaptation						
ReAct + ACE w/o Reflector or multi-epoch	✓	70.8$_{+7.1}$	55.4$_{+12.5}$	55.9$_{+14.4}$	38.1$_{+17.5}$	55.1$_{+12.7}$
ReAct + ACE w/o multi-epoch	✓	72.0$_{+8.3}$	60.7$_{+17.8}$	54.9$_{+13.4}$	39.6$_{+18.0}$	56.8$_{+14.4}$
ReAct + ACE	✓	76.2$_{+12.5}$	64.3$_{+21.4}$	57.3$_{+15.8}$	39.6$_{+18.0}$	59.4$_{+17.0}$
Online Adaptation						
ReAct + ACE	✗	67.9$_{+4.2}$	51.8$_{+8.9}$	61.4$_{+19.9}$	43.2$_{+21.6}$	56.1$_{+13.7}$
ReAct + ACE + offline warmup	✗	69.6$_{+5.9}$	53.6$_{+10.7}$	66.0$_{+24.5}$	48.9$_{+27.3}$	59.5$_{+17.1}$

Table 3: **Ablation Studies on AppWorld.** We study how particular design choices of ACE (iterative refinement, multi-epoch adaptation, and offline warmup) could help high-quality context adaptation.

Method	Latency (s)↓	# Rollouts↓
ReAct + GEPA	53898	1434
ReAct + ACE	9517$_{-82.3\%}$	357$_{-75.1\%}$

(a) **Offline** (AppWorld).

Method	Latency (s)↓	Token Cost ($)↓
DC (CU)	65104	17.7
ACE	5503$_{-91.5\%}$	2.9$_{-83.6\%}$

(b) **Online** (FiNER).

Figure 5: **Cost and Speed Analysis.** We measure the context adaptation latency, number of rollouts and dollar costs of ACE against GEPA (offline) and DC (online).

As examples, on the offline adaptation of AppWorld, ACE achieves 82.3% reduction in adaptation latency and 75.1% reduction in the number of rollouts as compared to GEPA (Figure 5a). On the online adaptation of FiNER, ACE achieves 91.5% reduction in adaptation latency and 83.6% reduction in token dollar cost for token ingestion/generation as compared to DC (Figure 5b).

5 Discussion

Longer Context ≠ Higher Serving Cost. Although ACE produces longer contexts than methods such as GEPA, this does not translate to linearly higher inference cost or GPU memory usage. Modern serving infrastructures are increasingly optimized for long-context workloads through techniques such as the reuse (Gim et al., 2024; Yao et al., 2025), compression (Liu et al., 2024b;a), and offload (Lee et al., 2024) of KV cache. These mechanisms allow frequently reused context segments to be cached locally or remotely, avoiding repetitive and expensive prefill operations. Ongoing advances in ML systems suggest that the amortized cost of handling long contexts will continue to decrease, making context-rich approaches like ACE increasingly practical in deployment.

Limitations and Challenges. A limitation of ACE is its reliance on a reasonably strong Reflector: if the Reflector fails to extract meaningful insights from generated traces or outcomes, the constructed context may become noisy or even harmful. In domain-specific tasks where no model can extract useful insights, the resulting context will naturally lack them. This dependency is similar to Dynamic Cheatsheet (Suzgun et al., 2025b), where the quality of adaptation hinges on the underlying model's ability to curate memory. We also note that not all applications require rich or detailed contexts. Tasks like HotPotQA (Yang et al., 2018) often benefit more from concise, high-level instructions (*e.g.,* how to retrieve and synthesize evidence) than from long contexts. Similarly, games with fixed strategies such as Game of 24 (Suzgun et al., 2025b) may only need a single reusable rule, rendering additional context redundant. Overall, ACE is most beneficial in settings that demand detailed domain knowledge, complex tool use, or environment-specific strategies that go beyond what is already embedded in model weights or simple system instructions.

ETHICS STATEMENT

This work does not raise specific ethical concerns. Our contributions focus on developing algorithms and system frameworks for effective context adaptation in large language models (LLMs). All experiments are conducted on publicly available benchmarks with open-source models, without involving human subjects, sensitive data, or privacy-related information. No potential conflicts of interest are present.

REPRODUCIBILITY STATEMENT

We provide detailed descriptions of our experimental setup, including datasets, benchmarks, evaluation metrics, baselines, and hyperparameter choices. Additional details, such as prompts for large language models and extended experimental settings, are included in the appendix. With this information, readers with reasonable computational resources should be able to reproduce our results.

REFERENCES

Rishabh Agarwal, Avi Singh, Lei Zhang, Bernd Bohnet, Luis Rosias, Stephanie Chan, Biao Zhang, Ankesh Anand, Zaheer Abbas, Azade Nova, et al. Many-shot in-context learning. *Advances in Neural Information Processing Systems*, 37:76930–76966, 2024.

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.

AppWorld. Leaderboard. `https://appworld.dev/leaderboard`, 2025. Accessed: 2025-09-24.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection. 2024.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pp. 2206–2240. PMLR, 2022.

Tianxiang Chen, Zhentao Tan, Xiaofan Bo, Yue Wu, Tao Gong, Qi Chu, Jieping Ye, and Nenghai Yu. Flora: Effortless context construction to arbitrary length and scale. *arXiv preprint arXiv:2507.19786*, 2025.

Yeounoh Chung, Gaurav T Kakkar, Yu Gan, Brenton Milne, and Fatma Ozcan. Is long context all you need? leveraging llm's extended context for nl2sql. *arXiv preprint arXiv:2501.12372*, 2025.

DeepSeek-AI. Deepseek-v3 technical report, 2024. URL `https://arxiv.org/abs/2412.19437`.

DSPy. dspy.gepa: Reflective prompt optimizer. `https://dspy.ai/api/optimizers/GEPA/overview/`, 2025a. Accessed: 2025-09-24.

DSPy. dspy.miprov2. `https://dspy.ai/api/optimizers/MIPROv2/`, 2025b. Accessed: 2025-09-24.

Shuzheng Gao, Chaozheng Wang, Cuiyun Gao, Xiaoqian Jiao, Chun Yong Chong, Shan Gao, and Michael Lyu. The prompt alchemist: Automated llm-tailored prompt optimization for test case generation. *arXiv preprint arXiv:2501.01329*, 2025.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

Neel Guha, Julian Nyarko, Daniel Ho, Christopher Ré, Adam Chilton, Alex Chohlas-Wood, Austin Peters, Brandon Waldon, Daniel Rockmore, Diego Zambrano, et al. Legalbench: A collaboratively built benchmark for measuring legal reasoning in large language models. *Advances in neural information processing systems*, 36:44123–44279, 2023.

Mingjian Jiang, Yangjun Ruan, Luis Lastras, Pavan Kapanipathi, and Tatsunori Hashimoto. Putting it all into context: Simplifying agents with lclms. *arXiv preprint arXiv:2505.08120*, 2025.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*, 2022.

Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. Dynamic evaluation of transformer language models. *arXiv preprint arXiv:1904.08378*, 2019.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 155–172, 2024.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33: 9459–9474, 2020.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 38–56, 2024a.

Zhining Liu, Rana Ali Amjad, Ravinarayana Adkathimar, Tianxin Wei, and Hanghang Tong. Selfelicit: Your language model secretly knows where is the relevant evidence. *arXiv preprint arXiv:2502.08767*, 2025.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024b.

Lefteris Loukas, Manos Fergadiotis, Ilias Chalkidis, Eirini Spyropoulou, Prodromos Malakasiotis, Ion Androutsopoulos, and Georgios Paliouras. Finer: Financial numeric entity recognition for xbrl tagging. *arXiv preprint arXiv:2203.06482*, 2022.

Yansheng Mao, Jiaqi Li, Fanxu Meng, Jing Xiong, Zilong Zheng, and Muhan Zhang. Lift: Improving long context understanding through long input fine-tuning. *arXiv preprint arXiv:2412.13626*, 2024.

Sami Marreed, Alon Oved, Avi Yaeli, Segev Shlomov, Ido Levy, Offer Akrabi, Aviad Sela, Asaf Adi, and Nir Mashkif. Towards enterprise-ready computer using generalist agent. *arXiv preprint arXiv:2503.01861*, 2025.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. *arXiv preprint arXiv:2406.11695*, 2024.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565, 2024.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*, 2023.

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

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. https://github.com/suzgunmirac/dynamic-cheatsheet, 2025a. Accessed: 2025-09-24.

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

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

Dannong Wang, Jaisal Patel, Daochen Zha, Steve Y Yang, and Xiao-Yang Liu. Finlora: Benchmarking lora methods for fine-tuning llms on financial datasets. *arXiv preprint arXiv:2505.19819*, 2025.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

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

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.

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

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*, 2018.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 94–109, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Jiacheng Ye, Chengzu Li, Lingpeng Kong, and Tao Yu. Generating data for symbolic language with large language models. *arXiv preprint arXiv:2305.13917*, 2023.

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.

Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. The shift from models to compound ai systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024.

Genghan Zhang, Weixin Liang, Olivia Hsu, and Kunle Olukotun. Adaptive self-improvement llm agentic system for ml library development. *arXiv preprint arXiv:2502.02534*, 2025a.

Qizheng Zhang, Ali Imran, Enkeleda Bardhi, Tushar Swamy, Nathan Zhang, Muhammad Shahbaz, and Kunle Olukotun. Caravan: Practical online learning of {In-Network}{ML} models with labeling agents. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 325–345, 2024.

Qizheng Zhang, Michael Wornow, and Kunle Olukotun. Cost-efficient serving of llm agents via test-time plan caching. *arXiv preprint arXiv:2506.14852*, 2025b.

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

A Related Work

A.1 Agent Memory

A growing body of work explores how agents can accumulate experience from past trajectories and leverage external (often non-parametric) memory to guide future actions. AgentFly (Zhou et al., 2025) presents an extensible framework where memory evolves continuously as agents solve tasks, enabling scalable reinforcement learning and long-horizon reasoning across diverse environments. AWM (Agent Workflow Memory) (Wang et al., 2024) induces reusable *workflows*—structured routines distilled from past trajectories—and selectively injects them into memory to improve efficiency and generalization in web navigation benchmarks. A-MEM (Xu et al., 2025) introduces a dynamically organized memory system inspired by the Zettelkasten method: each stored memory is annotated with structured attributes (*e.g.,* tags, keywords, contextual descriptions) and automatically linked to relevant past entries, while existing entries are updated to integrate new knowledge, yielding adaptive and context-aware retrieval. Agentic Plan Caching (Zhang et al., 2025b) instead focuses on cost efficiency by extracting reusable plan templates from agent trajectories and caching them for fast execution at test time.

Together, these works demonstrate the value of external memory for improving adaptability, efficiency, and generalization in LLM agents. Our work differs by tackling the broader challenge of *context adaptation*, which spans not only agent memory but also system prompts, factual evidence, and other inputs underpinning AI systems. We further highlight two fundamental limitations of existing adaptation methods—*brevity bias* and *context collapse*—and show that addressing them is essential for robustness, reliability, and scalability beyond raw task performance. Accordingly, our evaluation considers not only accuracy but also cost, latency, and scalability.

B The Use of Large Language Models (LLMs)

This work focuses on developing algorithms and system frameworks for effective context adaptation in large language models (LLMs). Accordingly, our experiments employ LLMs for the empirical evaluation of the proposed methods. For paper preparation, we used LLMs only to polish writing (*e.g.,* correcting grammatical errors), and not to generate new text from scratch.

C PROMPTS

I am your supervisor and you are a super intelligent AI Assistant whose job is to achieve my day-to-day tasks completely autonomously.

To do this, you will need to interact with app/s (e.g., spotify, venmo etc) using their associated APIs on my behalf. For this you will undertake a *multi-step conversation* using a python REPL environment. That is, you will write the python code and the environment will execute it and show you the result, based on which, you will write python code for the next step and so on, until you've achieved the goal. This environment will let you interact with app/s using their associated APIs on my behalf.

Here are three key APIs that you need to know to get more information

```
# To get a list of apps that are available to you.
print(apis.api_docs.show_app_descriptions())

# To get the list of apis under any app listed above, e.g. spotify
print(apis.api_docs.show_api_descriptions(app_name='spotify'))

# To get the specification of a particular api, e.g. spotify app's login api
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='login'))
```

Each code execution will produce an output that you can use in subsequent calls. Using these APIs, you can now generate code, that I will execute, to solve the task.

Let's start with the task

[3 shot example]

Key instructions:

1. Make sure to end code blocks with ``` followed by a newline().
2. Remember you can use the variables in your code in subsequent code blocks.
3. Remember that the email addresses, access tokens and variables (e.g. spotify_password) in the example above are not valid anymore.
4. You can use the "supervisor" app to get information about my accounts and use the "phone" app to get information about friends and family.
5. Always look at API specifications (using `apis.api_docs.show_api_doc`) before calling an API.
6. Write small chunks of code and only one chunk of code in every step. Make sure everything is working correctly before making any irreversible change.
7. Many APIs return items in "pages". Make sure to run through all the pages by looping over `page_index`.
8. Once you have completed the task, make sure to call `apis.supervisor.complete_task()`. If the task asked for some information, return it as the answer argument, i.e. call `apis.supervisor.complete_task(answer=<answer>)`. Many tasks do not require an answer, so in those cases, just call `apis.supervisor.complete_task()` i.e. do not pass any argument.

Using these APIs, generate code to solve the actual task:

My name is: {{ main_user.first_name }} {{ main_user.last_name }}. My personal email is {{ main_user.email }} and phone number is {{ main_user.phone_number }}.

Task: {{ input_str }}

Figure 6: ICL-baseline Generator prompt on AppWorld

I am your supervisor and you are a super intelligent AI Assistant whose job is to achieve my day-to-day tasks completely autonomously. You will be given a cheatsheet containing relevant strategies, patterns, and examples from similar problems to apply and solve the current task.

To do this, you will need to interact with app/s (e.g., spotify, venmo etc) using their associated APIs on my behalf. For this you will undertake a *multi-step conversation* using a python REPL environment. That is, you will write the python code and the environment will execute it and show you the result, based on which, you will write python code for the next step and so on, until you've achieved the goal. This environment will let you interact with app/s using their associated APIs on my behalf.

Here are three key APIs that you need to know to get more information

```
# To get a list of apps that are available to you.
print(apis.api_docs.show_app_descriptions())

# To get the list of apis under any app listed above, e.g. spotify
print(apis.api_docs.show_api_descriptions(app_name='spotify'))

# To get the specification of a particular api, e.g. spotify app's login api
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='login'))
```

Each code execution will produce an output that you can use in subsequent calls. Using these APIs, you can now generate code, that I will execute, to solve the task.

CHEATSHEET: ''' {{ cheat_sheet }} '''

---

1. ANALYSIS & STRATEGY

- Carefully analyze both the question and cheatsheet before starting
- Search for and identify any applicable patterns, strategies, or examples within the cheatsheet
- Create a structured approach to solving the problem at hand
- Review and document any limitations in the provided reference materials

2. SOLUTION DEVELOPMENT

- Present your solution using clear, logical steps that others can follow and review
- Explain your reasoning and methodology before presenting final conclusions
- Provide detailed explanations for each step of the process
- Check and verify all assumptions and intermediate calculations

3. PROGRAMMING TASKS

When coding is required: - Write clean, efficient Python code - Follow the strict code formatting and execution protocol (always use the Python code formatting block; furthermore, after the code block, always explicitly request execution by appending: "EXECUTE CODE!"): `python # Your code here` EXECUTE CODE!

- All required imports and dependencies should be clearly declared at the top of your code
- Include clear inline comments to explain any complex programming logic
- Perform result validation after executing your code
- Apply optimization techniques from the cheatsheet when applicable
- The code should be completely self-contained without external file dependencies–it should be ready to be executed right away
- Do not include any placeholders, system-specific paths, or hard-coded local paths
- Feel free to use standard and widely-used pip packages
- Opt for alternative methods if errors persist during execution
- Exclude local paths and engine-specific settings (e.g., avoid configurations like chess.engine.SimpleEngine.popen_uci("/usr/bin/stockfish"))

Let's start with the task

[3 shot example]

Key instructions: (1) Make sure to end code blocks with ``` followed by a newline().

2. Remember you can use the variables in your code in subsequent code blocks.
3. Remember that the email addresses, access tokens and variables (e.g. spotify_password) in the example above are not valid anymore.
4. You can use the "supervisor" app to get information about my accounts and use the "phone" app to get information about friends and family.
5. Always look at API specifications (using `apis.api_docs.show_api_doc`) before calling an API.
6. Write small chunks of code and only one chunk of code in every step. Make sure everything is working correctly before making any irreversible change.
7. Many APIs return items in "pages". Make sure to run through all the pages by looping over `page_index`.
8. Once you have completed the task, make sure to call `apis.supervisor.complete_task()`. If the task asked for some information, return it as the answer argument, i.e. call `apis.supervisor.complete_task(answer=<answer>)`. Many tasks do not require an answer, so in those cases, just call `apis.supervisor.complete_task()` i.e. do not pass any argument.

Using these APIs, generate code to solve the actual task:

My name is: {{ main_user.first_name }} {{ main_user.last_name }}. My personal email is {{ main_user.email }} and phone number is {{ main_user.phone_number }}. Task: {{ input_str }}

Figure 7: Dynamic Cheatsheet Generator prompt on AppWorld

I am your supervisor and you are a super intelligent AI Assistant whose job is to achieve my day-to-day tasks completely autonomously.

To do this, you will need to interact with app/s (e.g., spotify, venmo etc) using their associated APIs on my behalf. For this you will undertake a *multi-step conversation* using a python REPL environment. That is, you will write the python code and the environment will execute it and show you the result, based on which, you will write python code for the next step and so on, until you've achieved the goal. This environment will let you interact with app/s using their associated APIs on my behalf.

Here are three key APIs that you need to know to get more information:

```
# To get a list of apps that are available to you.
print(apis.api_docs.show_app_descriptions())

# To get the list of apis under any app listed above, e.g. spotify
print(apis.api_docs.show_api_descriptions(app_name='spotify'))

# To get the specification of a particular api, e.g. spotify app's login api
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='login'))
```

Each code execution will produce an output that you can use in subsequent calls. Using these APIs, you can now generate code, that I will execute, to solve the task.

Key Instructions:

1. Always end code blocks with ``` followed by a newline().
2. Remember you can use variables in your code in subsequent code blocks.
3. Email addresses, access tokens and variables from previous examples are not valid anymore.
4. Use the "supervisor" app to get information about my accounts and the "phone" app to get information about friends and family.
5. Always look at API specifications (using `apis.api_docs.show_api_doc`) before calling an API.
6. Write small chunks of code and only one chunk of code in every step. Make sure everything is working correctly before making any irreversible changes.
7. Many APIs return items in "pages". Make sure to run through all the pages by looping over `page_index`.
8. Once you have completed the task, call `apis.supervisor.complete_task()`. If the task asked for information, return it as the answer argument: `apis.supervisor.complete_task(answer=<answer>)`. For tasks without required answers, just call `apis.supervisor.complete_task()` without arguments.

Domain-Specific Strategy for Bill Splitting Tasks: When splitting bills among roommates, remember to: - First identify roommates using phone app's search_contacts with "roommate" relationship query - Access bill receipts in file system under "/home/[username]/bills/" directory structure - Calculate equal shares by dividing total amount by (number of roommates + 1) including yourself - Use Venmo's create_payment_request API with roommates' email addresses - Ensure payment requests are only sent to actual roommates (not coworkers or other contacts) - Verify that all roommates have the same home address in their contact information - Use the description "I paid for cable bill." for payment requests

Domain-Specific Strategy for File Organization Tasks: When organizing files based on creation dates, remember to: - First login to the file system using credentials from supervisor - Use show_directory() to list files and show_file() to get file metadata including created_at - Create destination directories using create_directory() before moving files - Use move_file() to organize files while maintaining original filenames - Files created in specific months should be moved to corresponding destination directories (e.g., March → Rome, April → Santorini, others → Berlin)

Domain-Specific Strategy for Music Playlist Tasks: When creating playlists for specific durations, remember to: - Calculate total duration needed (e.g., 90 minutes = 5400 seconds) - Search for appropriate songs across different genres (workout, energetic, rock, pop, dance) - Use show_song() to get individual song durations - Add songs to playlist until total duration requirement is met - Use play_music() with playlist_id to start playback

Domain-Specific Strategy for File Compression Tasks: When compressing vacation photo directories, remember to: - Compress each vacation spot directory individually - Save compressed files in the specified destination path format (e.g., "~/photographs/vacations/.zip") - Delete the original directories after successful compression - Verify that the compressed files are created in the correct location

Domain-Specific Strategy for Alarm Management Tasks: When modifying phone alarms, remember to: - Identify the specific alarm by its label (e.g., "Wake Up") - Calculate new times accurately (convert HH:MM to minutes for arithmetic operations) - Disable all other enabled alarms except the one being modified - Preserve all other alarm settings while making changes

Domain-Specific Strategy for Message Management Tasks: When handling text/voice messages, remember to: - Use search functions to find specific messages by phone number or content - Handle pagination to ensure all relevant messages are processed - Delete messages using their specific message IDs - Verify deletion by checking that no messages remain

Let's start with the task:

Figure 8: GEPA prompt on AppWorld

I am your supervisor and you are a super intelligent AI Assistant whose job is to achieve my day-to-day tasks completely autonomously.

To do this, you will need to interact with app/s (e.g., spotify, venmo etc) using their associated APIs on my behalf. For this you will undertake a *multi-step conversation* using a python REPL environment. That is, you will write the python code and the environment will execute it and show you the result, based on which, you will write python code for the next step and so on, until you've achieved the goal. This environment will let you interact with app/s using their associated APIs on my behalf.

Here are three key APIs that you need to know to get more information

```
# To get a list of apps that are available to you.
print(apis.api_docs.show_app_descriptions())

# To get the list of apis under any app listed above, e.g. spotify
print(apis.api_docs.show_api_descriptions(app_name='spotify'))

# To get the specification of a particular api, e.g. spotify app's login api
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='login'))
```

Each code execution will produce an output that you can use in subsequent calls. Using these APIs, you can now generate code, that I will execute, to solve the task.

You are also provided with a curated cheatsheet of strategies, API-specific information, common mistakes, and proven solutions to help you solve the task effectively.

ACE Playbook: - Read the **Playbook** first, then execute the task by explicitly leveraging each relevant section:

PLAYBOOK_BEGIN

{{ playbook }}

PLAYBOOK_END

Let's start with the task

[3 shot example]

Key instructions:

1. Make sure to end code blocks with ``` followed by a newline().
2. Remember you can use the variables in your code in subsequent code blocks.
3. Remember that the email addresses, access tokens and variables (e.g. spotify_password) in the example above are not valid anymore.
4. You can use the "supervisor" app to get information about my accounts and use the "phone" app to get information about friends and family.
5. Always look at API specifications (using `apis.api_docs.show_api_doc`) before calling an API.
6. Write small chunks of code and only one chunk of code in every step. Make sure everything is working correctly before making any irreversible change.
7. Many APIs return items in "pages". Make sure to run through all the pages by looping over `page_index`.
8. Once you have completed the task, make sure to call `apis.supervisor.complete_task()`. If the task asked for some information, return it as the answer argument, i.e. call `apis.supervisor.complete_task(answer=<answer>)`. Many tasks do not require an answer, so in those cases, just call `apis.supervisor.complete_task()` i.e. do not pass any argument.
9. Treat the cheatsheet as a tool. Use only the parts that are relevant and applicable to your specific situation and task context, otherwise use your own judgement.

Using these APIs and cheatsheet, generate code to solve the actual task:

My name is: {{ main_user.first_name }} {{ main_user.last_name }}. My personal email is {{ main_user.email }} and phone number is {{ main_user.phone_number }}. Task: {{ input_str }}

Figure 9: ACE Generator prompt on AppWorld

You are an expert AppWorld coding agent and educator. Your job is to diagnose the current trajectory: identify what went wrong (or could be better), grounded in execution feedback, API usage, unit test report, and ground truth when applicable.

Instructions: - Carefully analyze the model's reasoning trace to identify where it went wrong - Take the environment feedback into account, comparing the predicted answer with the ground truth to understand the gap - Identify specific conceptual errors, calculation mistakes, or misapplied strategies - Provide actionable insights that could help the model avoid this mistake in the future - Identify root causes: wrong source of truth, bad filters (timeframe/direction/identity), formatting issues, or missing authentication and how to correct them. - Provide concrete, step-by-step corrections the model should take in this task. - Be specific about what the model should have done differently - You will receive bulletpoints that are part of playbook that's used by the generator to answer the question. - You need to analyze these bulletpoints, and give the tag for each bulletpoint, tag can be ['helpful', 'harmful', 'neutral'] (for the generator to generate the correct answer) - Explicitly curate from the environment feedback the output format/schema of APIs used when unclear or mismatched with expectations (e.g., `apis.blah.show_contents()` returns a list of content_ids (strings), not content objects)

Inputs:

- Ground truth code (reference, known-correct):

GROUND_TRUTH_CODE_START

{{ground_truth_code}}

GROUND_TRUTH_CODE_END

- Test report (unit tests result for the task after the generated code was run):

TEST_REPORT_START

{{unit_test_results}}

TEST_REPORT_END

- ACE playbook (playbook that's used by model for code generation):

PLAYBOOK_START

{{playbook}}

PLAYBOOK_END

Examples:

Example 1:

Ground Truth Code: [Code that uses apis.phone.search_contacts() to find roommates, then filters Venmo transactions]

Generated Code: [Code that tries to identify roommates by parsing Venmo transaction descriptions using keywords like "rent", "utilities"]

Execution Error: AssertionError: Expected 1068.0 but got 79.0

Test Report: FAILED - Wrong total amount calculated due to incorrect roommate identification

Response:

{{

"reasoning": "The generated code attempted to identify roommates by parsing Venmo transaction descriptions rather than using the authoritative Phone app contacts. This led to missing most roommate transactions and calculating an incorrect total of 79.0 instead of 1068.0.",

"error_identification": "The agent used unreliable heuristics (keyword matching in transaction descriptions) to identify roommates instead of the correct API (Phone contacts).",

"root_cause_analysis": "The agent misunderstood the data architecture - it assumed transaction descriptions contained reliable relationship information, when the Phone app is the authoritative source for contact relationships.",

"correct_approach": "First authenticate with Phone app, use apis.phone.search_contacts() to identify contacts with 'roommate' relationship, then filter Venmo transactions by those specific contact emails/phone numbers.",

"key_insight": "Always resolve identities from the correct source app - Phone app for relationships, never rely on transaction descriptions or other indirect heuristics which are unreliable."

}}

Example 2:

Ground Truth Code: [Code that uses proper while True pagination loop to get all Spotify playlists]

Generated Code: [Code that uses for i in range(10) to paginate through playlists]

Execution Error: None (code ran successfully)

Test Report: FAILED - Expected 23 playlists but got 10 due to incomplete pagination

Response:

{{

"reasoning": "The generated code used a fixed range loop (range(10)) for pagination instead of properly iterating until no more results are returned. This caused the agent to only collect the first 10 pages of playlists, missing 13 additional playlists that existed on later pages.",

"error_identification": "The pagination logic used an arbitrary fixed limit instead of continuing until all pages were processed.",

"root_cause_analysis": "The agent used a cautious approach with a fixed upper bound to avoid infinite loops, but this prevented complete data collection when the actual data exceeded the arbitrary limit.",

"correct_approach": "Use while True loop with proper break condition: continue calling the API with incrementing page_index until the API returns empty results or null, then break.",

"key_insight": "For pagination, always use while True loop instead of fixed range iterations to ensure complete data collection across all available pages."

}}

Outputs: Your output should be a json object, which contains the following fields - reasoning: your chain of thought / reasoning / thinking process, detailed analysis and calculations - error_identification: what specifically went wrong in the reasoning? - root_cause_analysis: why did this error occur? What concept was misunderstood? - correct_approach: what should the model have done instead? - key_insight: what strategy, formula, or principle should be remembered to avoid this error?

Answer in this exact JSON format:

{{

"reasoning": "[Your chain of thought / reasoning / thinking process, detailed analysis and calculations]",

"error_identification": "[What specifically went wrong in the reasoning?]",

"root_cause_analysis": "[Why did this error occur? What concept was misunderstood?]",

"correct_approach": "[What should the model have done instead?]",

"key_insight": "[What strategy, formula, or principle should be remembered to avoid this error?]",

}}

[FULL AGENT-ENVIRONMENT TRAJECTORY ATTACHED HERE]

Figure 10: ACE Reflector prompt on AppWorld

You are a master curator of knowledge. Your job is to identify what new insights should be added to an existing playbook based on a reflection from a previous attempt.

Context: - The playbook you created will be used to help answering similar questions. - The reflection is generated using ground truth answers that will NOT be available when the playbook is being used. So you need to come up with content that can aid the playbook user to create predictions that likely align with ground truth.

Instructions: - Review the existing playbook and the reflection from the previous attempt - Identify ONLY the NEW insights, strategies, or mistakes that are MISSING from the current playbook - Avoid redundancy - if similar advice already exists, only add new content that is a perfect complement to the existing playbook - Do NOT regenerate the entire playbook - only provide the additions needed - Focus on quality over quantity - a focused, well-organized playbook is better than an exhaustive one - Format your response as a PURE JSON object with specific sections - For any operation if no new content to add, return an empty list for the operations field - Be concise and specific - each addition should be actionable - For coding tasks, explicitly curate from the reflections the output format/schema of APIs used when unclear or mismatched with expectations (e.g., `apis.blah.show_contents()` returns a list of content_ids (strings), not content objects)

- **Task Context (the actual task instruction):**
  `{question_context}`
- **Current Playbook:**
  `{current_playbook}`
- **Current Generated Attempt (latest attempt, with reasoning and planning):**
  `{final_generated_code}`
- **Current Reflections (principles and strategies that helped to achieve current task):**
  `{guidebook}`

Examples:

Example 1:

Task Context: “Find money sent to roommates since Jan 1 this year”

Current Playbook: [Basic API usage guidelines]

Generated Attempt: [Code that failed because it used transaction descriptions to identify roommates instead of Phone contacts]

Reflections: “The agent failed because it tried to identify roommates by parsing Venmo transaction descriptions instead of using the Phone app’s contact relationships. This led to incorrect identification and wrong results.”

Response:

```
{
  "reasoning": "The reflection shows a critical error where the agent used unreliable heuristics (transaction descriptions) instead of the
       authoritative source (Phone app contacts) to identify relationships. This is a fundamental principle that should be captured in the
       playbook to prevent similar failures in identity resolution tasks.",
  "operations": [
    {
      "type": "ADD",
      "section": "strategies_and_hard_rules",
      "content": "Always resolve identities from the correct source app\n- When you need to identify relationships (roommates, contacts, etc.),
        always use the Phone app's contact, and never try other heuristics from transaction descriptions, name patterns, or other indirect
        sources. These heuristics are unreliable and will cause incorrect results."
    }
  ]
}
```

Example 2:

Task Context: “Count all playlists in Spotify”

Current Playbook: [Basic authentication and API calling guidelines]

Generated Attempt: [Code that used for i in range(10) loop and missed playlists on later pages]

Reflections: “The agent used a fixed range loop for pagination instead of properly iterating through all pages until no more results are returned. This caused incomplete data collection.”

Response:

```
{
  "reasoning": "The reflection identifies a pagination handling error where the agent used an arbitrary fixed range instead of proper pagination
       logic. This is a common API usage pattern that should be explicitly documented to ensure complete data retrieval.",
  "operations": [
    {
      "type": "ADD",
      "section": "apis_to_use_for_specific_information",
      "content": "About pagination: many APIs return items in \"pages\". Make sure to run through all the pages using while True loop instead of
        for i in range(10) over `page_index`."
    }
  ]
}
```

Your Task: Output ONLY a valid JSON object with these exact fields: - reasoning: your chain of thought / reasoning / thinking process, detailed analysis and calculations - operations: a list of operations to be performed on the playbook - type: the type of operation to be performed - section: the section to add the bullet to - content: the new content of the bullet

Available Operations: 1. ADD: Create new bullet points with fresh IDs - section: the section to add the new bullet to - content: the new content of the bullet. Note: no need to include the bullet_id in the content like ‘[ctx-00263] helpful=1 harmful=0 ::’, the bullet_id will be added by the system.

RESPONSE FORMAT - Output ONLY this JSON structure (no markdown, no code blocks):

```
{
  "reasoning": "[Your chain of thought / reasoning / thinking process, detailed analysis and calculations here]",
  "operations": [
    {
      "type": "ADD",
      "section": "verification_checklist",
      "content": "[New checklist item or API schema clarification...]"
    }
  ]
}
```

Figure 11: ACE Curator prompt on AppWorld

You are an analysis expert tasked with answering questions using your knowledge, a curated playbook of strategies and insights and a reflection that goes over the diagnosis of all previous mistakes made while answering the question.

Instructions: - Read the playbook carefully and apply relevant strategies, formulas, and insights - Pay attention to common mistakes listed in the playbook and avoid them - Show your reasoning step-by-step - Be concise but thorough in your analysis - If the playbook contains relevant code snippets or formulas, use them appropriately - Double-check your calculations and logic before providing the final answer

Your output should be a json object, which contains the following fields: - reasoning: your chain of thought / reasoning / thinking process, detailed analysis and calculations - bullet_ids: each line in the playbook has a bullet_id. all bulletpoints in the playbook that's relevant, helpful for you to answer this question, you should include their bullet_id in this list - final_answer: your concise final answer

Playbook:

{}

Reflection:

{}

Question:

{}

Context:

{}

Answer in this exact JSON format:

```
{
  "reasoning": "[Your chain of thought / reasoning / thinking process, detailed analysis and calculations]",
  "bullet_ids": ["calc-00001", "fin-00002"],
  "final_answer": "[Your concise final answer here]"
}
```

Figure 12: ACE Generator prompt on FINER

You are an expert analyst and educator. Your job is to diagnose why a model's reasoning went wrong by analyzing the gap between predicted answer and the ground truth.

Instructions: - Carefully analyze the model's reasoning trace to identify where it went wrong - Take the environment feedback into account, comparing the predicted answer with the ground truth to understand the gap - Identify specific conceptual errors, calculation mistakes, or misapplied strategies - Provide actionable insights that could help the model avoid this mistake in the future - Focus on the root cause, not just surface-level errors - Be specific about what the model should have done differently - You will receive bulletpoints that are part of playbook that's used by the generator to answer the question. - You need to analyze these bulletpoints, and give the tag for each bulletpoint, tag can be ['helpful', 'harmful', 'neutral'] (for the generator to generate the correct answer)

Your output should be a json object, which contains the following fields - reasoning: your chain of thought / reasoning / thinking process, detailed analysis and calculations - error_identification: what specifically went wrong in the reasoning? - root_cause_analysis: why did this error occur? What concept was misunderstood? - correct_approach: what should the model have done instead? - key_insight: what strategy, formula, or principle should be remembered to avoid this error? - bullet_tags: a list of json objects with bullet_id and tag for each bulletpoint used by the generator

Question:

{}

Model's Reasoning Trace:

{}

Model's Predicted Answer:

{}

Ground Truth Answer:

{}

Environment Feedback:

{}

Part of Playbook that's used by the generator to answer the question:

{}

Answer in this exact JSON format:

```
{
  "reasoning": "[Your chain of thought / reasoning / thinking process, detailed analysis and calculations]",
  "error_identification": "[What specifically went wrong in the reasoning?]",
  "root_cause_analysis": "[Why did this error occur? What concept was misunderstood?]",
  "correct_approach": "[What should the model have done instead?]",
  "key_insight": "[What strategy, formula, or principle should be remembered to avoid this error?]",
  "bullet_tags": [
    {{"id": "calc-00001", "tag": "helpful"}},
    {{"id": "fin-00002", "tag": "harmful"}}
  ]
}
```

Figure 13: ACE Reflector prompt on FINER

You are a master curator of knowledge. Your job is to identify what new insights should be added to an existing playbook based on a reflection from a previous attempt.

Context: - The playbook you created will be used to help answering similar questions. - The reflection is generated using ground truth answers that will NOT be available when the playbook is being used. So you need to come up with content that can aid the playbook user to create predictions that likely align with ground truth.

CRITICAL: You MUST respond with valid JSON only. Do not use markdown formatting or code blocks.

Instructions: - Review the existing playbook and the reflection from the previous attempt - Identify ONLY the NEW insights, strategies, or mistakes that are MISSING from the current playbook - Avoid redundancy - if similar advice already exists, only add new content that is a perfect complement to the existing playbook - Do NOT regenerate the entire playbook - only provide the additions needed - Focus on quality over quantity - a focused, well-organized playbook is better than an exhaustive one - Format your response as a PURE JSON object with specific sections - For any operation if no new content to add, return an empty list for the operations field - Be concise and specific - each addition should be actionable

Training Context:

- Total token budget: {token_budget} tokens
- Training progress: Sample {current_step} out of {total_samples}

Current Playbook Stats:

{playbook_stats}

Recent Reflection:

{recent_reflection}

Current Playbook:

{current_playbook}

Question Context:

{question_context}

Your Task: Output ONLY a valid JSON object with these exact fields: - reasoning: your chain of thought / reasoning / thinking process, detailed analysis and calculations - operations: a list of operations to be performed on the playbook - type: the type of operation to be performed - section: the section to add the bullet to - content: the new content of the bullet

Available Operations: 1. ADD: Create new bullet points with fresh IDs - section: the section to add the new bullet to - content: the new content of the bullet. Note: no need to include the bullet_id in the content like '[ctx-00263] helpful=1 harmful=0 ::', the bullet_id will be added by the system.

RESPONSE FORMAT - Output ONLY this JSON structure (no markdown, no code blocks):

```
{
  "reasoning": "[Your chain of thought / reasoning / thinking process, detailed analysis and calculations here]",
  "operations": [
    {{
      "type": "ADD",
      "section": "formulas_and_calculations",
      "content": "[New calculation method...]"
    }}
  ]
}
```

Figure 14: ACE Curator prompt on FINER