LEARNING TO MANAGE CONTEXT: TRAINING AGENTS WITH META-TOOLS BEYOND A SINGLE CONTEXT WINDOW

Anonymous authors
Paper under double-blind review

ABSTRACT

An agent that continually accumulates interaction history eventually exhausts its context window, thereby limiting the task horizons over which they can be trained. We introduce the Concurrent Agent Rollout Process (CARP), which brings context-management mechanisms into a common framework for agent execution and training. CARP encapsulates these mechanisms as meta-tools, allowing agents to replace accumulated histories with summaries or delegate work to other agent instances. By retaining each generated token's conditioning context, we can compute its log-probability for gradient-based training without requiring the complete execution to fit within one context window. We address obstacles which prevent agents from reasoning about context capacity. Using Group Relative Policy Optimisation (GRPO), agents learn to solve synthetic counting tasks whose data exceeds their context window, making context management necessary. Different reward and prompt conditions produce different strategies, including parallel task decomposition when the reward accounts for parallel computation savings. These results lay the groundwork for end-to-end training of agents on tasks whose execution extends beyond a fixed context budget.

1 INTRODUCTION

Language-model agents solve tasks through repeated interactions with an external environment, interleaving reasoning with tool calls and observations (Yao et al., 2023). If each interaction is appended to the history supplied to the model, that history eventually exceeds its context window. A training setup that assumes the complete interaction history fits within a single context window therefore limits the length of task executions it can train on. One way to extend training beyond this limit is to let the agent modify its context during execution, while retaining the information needed to train on its earlier actions. We present a framework for this approach, in which agents learn to manage their own context as part of solving the task.

Memory management offers a way to sustain interactions beyond a fixed context window (Packer et al., 2023). For example, an agent could replace its accumulated interaction history with a shorter summary and continue from that summary, an operation we call *compaction*. It could also start another instance of the same agent with a separate history, give it a task, and receive its answer. We call this operation *delegation*, and the new instance a *subagent*. A summary may omit a necessary fact, while a subagent may receive too little information to complete its task. These operations also change the training record: later outputs depend on the summary or the subagent's history, rather than on all preceding interactions. Training must preserve the context that produced each output while assigning credit for the outcome of the complete execution.

We introduce the *Concurrent Agent Rollout Process (CARP)*, a process for running an agent as a dynamically changing set of agent instances. It exposes *meta-tools*: tools that modify the agent's own execution state. This formalism encapsulates mechanisms such as compaction and subagents as operations on execution state: compaction replaces an instance's history, and delegation creates another instance. Both operations are available through the same tool-calling interface as actions on the external environment. The policy therefore chooses when to manage context, what state to carry forward, and what work to delegate, with these choices trained from the task outcome.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

The training construction rests on a simple observation: evaluating the log-probability of a sampled token requires the context under which it was generated. Retaining these contexts allows an episode's policy log-likelihood to be evaluated across compactions and subagents, without concatenating the whole execution into one sequence. We organise the saved contexts into token trees that record shared prefixes and sampled outputs. Their log-probability gradients can be evaluated in separate passes and accumulated, so the total execution need not fit in a single accelerator-resident sequence. This construction supports gradient-based objectives built from these token log-probabilities; each model call remains bounded by the context window. We also measure the total token cost across all threads and the cost along the longest chain of dependent work, which we call the *critical path*. These measures let us price both the amount of work performed and the extent to which it must occur sequentially.

A trainable representation alone does not ensure effective context management. In our experimental setting, a policy with access to both meta-tools still frequently exhausts its context before completing the task. We provide explicit context-usage counters and a coached prompt explaining the tools and asking the agent to consider its remaining capacity. We then train with group relative policy optimisation (GRPO; Shao et al., 2024), applying the episode-level training signal to generated tokens across all threads. This lets task outcomes supervise both the work performed and the decisions that determine its execution.

We first use a simple synthetic table-counting task, with files slightly larger than the 4,096-token context window, to examine whether training produces effective meta-tool use. Across four runs, best observed held-out success rates reach 48–84%, from 5% before training. To test usefulness beyond this exercise, we evaluate on OOLONG-synth, whose questions require interpreting and aggregating information across many text entries (Bertsch et al., 2025). On a subset selected to reduce simple answer priors, the counting-trained policy improves zero-shot success from 5% to 13% and solves 9 of 144 over-window episodes where the base model solves none. These results demonstrate learning and transfer within the framework, while leaving substantial limitations: success declines at longer input lengths, and the current two-seed comparison does not establish a benefit from the computation-cost reward.

Our contributions are:

- We introduce the Concurrent Agent Rollout Process (CARP), a process for running an agent as a dynamically changing set of agent instances.
- We introduce meta-tools that modify the state of this process, encapsulating mechanisms such as compaction and subagents.
- We establish that agents using CARP and meta-tools can be trained end-to-end with gradient-based objectives, without requiring the complete execution to fit within a single context window.
- We identify and address obstacles to learning context management through reinforcement learning.
- We define computation costs within CARP and incorporate them into the reinforcement-learning objective.
- We validate the framework by showing that reinforcement learning, without demonstrations of context-management strategies, enables agents to solve counting tasks whose inputs exceed their context window, and that the resulting policy transfers to OOLONG.

Together, these contributions lay the groundwork for end-to-end training agents on tasks whose execution extends beyond a single context window.

2 BACKGROUND

An autoregressive language model (LLM) with parameters θ defines a distribution over output token sequences $\mathbf{y}$ conditioned on an input context $\mathbf{x}$:

$$p_\theta(\mathbf{y} \mid \mathbf{x}) = \prod_{t=1}^{|\mathbf{y}|} p_\theta(y_t \mid \mathbf{x}, y_{<t}).$$

Consequently, the sequence log-likelihood decomposes into token-level log-probabilities,

$$\log p_\theta(\mathbf{y} \mid \mathbf{x}) = \sum_{t=1}^{|\mathbf{y}|} \log p_\theta(y_t \mid \mathbf{x}, y_{<t}). \tag{1}$$

A broad class of LLM post-training methods can be expressed in terms of gradients of token log-probabilities. In supervised fine-tuning (SFT), for example, the objective is the negative expected sequence log-likelihood,

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}} \left[\sum_{t=1}^{|\mathbf{y}|} \log p_\theta(y_t \mid \mathbf{x}, y_{<t}) \right].$$

More generally, policy-gradient methods weight these log-probability gradients according to the estimated utility of sampled responses. Abstracting away the particular estimator, their gradient takes the form

$$\nabla_\theta J(\theta) = \mathbb{E} \left[\sum_{t=1}^{|\mathbf{y}|} A_t \, \nabla_\theta \log p_\theta(y_t \mid \mathbf{x}, y_{<t}) \right],$$

where A_t denotes an advantage estimate. Practical objectives may additionally incorporate importance-sampling corrections, clipping, and regularization. Preference-based objectives such as DPO are similarly defined through log-probabilities assigned to alternative responses.

Crucially, although the weight assigned to a token may depend on information from the broader response or episode, the token's log-probability is always defined with respect to the context under which that token was generated. Writing this context as c_t, the corresponding term has the form

$$w_t \, \nabla_\theta \log p_\theta(y_t \mid c_t),$$

where w_t may encode, for example, an advantage, importance weight, or other training signal that is not itself local to c_t. Thus, it is the log-probability term, rather than the training objective as a whole, that is local to the token's conditioning context.

Agentic LLMs. An agent interacts with an external environment through repeated model calls. It uses *tools*, functions it calls to observe or act on that environment. Let h_j denote its structured message history at call j, including instructions, previous assistant messages, tool calls, and tool results. The call consists of encoding this history, generating tokens, and decoding them into an assistant message:

$$\begin{aligned} \mathbf{x}^{(j)} &= \mathrm{Encode}(h_j), \\ \mathbf{y}^{(j)} &\sim p_\theta(\cdot \mid \mathbf{x}^{(j)}), \\ m_j &= \mathrm{Decode}(\mathbf{y}^{(j)}). \end{aligned}$$

Encode formats and tokenises the history; Decode interprets the generated tokens as a message, including any tool calls. In a sequential interaction loop, as in ReAct (Yao et al., 2023), the agent appends m_j to the history, executes any requested tools, and appends their results to form h_{j+1}. Repeating this loop until termination, a process we call a *rollout*, produces a completed message history h_T, including the final assistant response, which we call an *episode*.

Training on agent interactions. Let P_θ denote the distribution over episodes induced by the rollout process. In the common case where $\mathbf{u} = \mathrm{Encode}(h_T)$ preserves the generated tokens and their conditioning contexts, the episode's log-probability gradient depends only on model-generated tokens, assuming the process supplying external inputs has no direct dependence on θ:

$$\nabla_\theta \log P_\theta(h_T) = \sum_{t=1}^{|\mathbf{u}|} b_t \, \nabla_\theta \log p_\theta(u_t \mid u_{<t}). \tag{2}$$

where b_t selects model-generated tokens. If $\mathbf{u}$ fits within the context window, one masked forward pass evaluates the log-probabilities; external tokens remain visible as context. The resulting log-probabilities and their gradients can therefore be used in the gradient-based optimisation methods described above.

3 Related Work

Learned memory and context management. Explicit memory management has long provided a way to operate beyond a fixed context window. MemGPT organises memory into tiers and gives the agent tools to move information between them (Packer et al., 2023). Subsequent work learns how to retain useful information: MEM1 integrates memory consolidation with reasoning through reinforcement learning (Zhou et al., 2026), while AgentFold learns proactive, multi-scale history folding from demonstrations (Ye et al., 2026). These approaches establish learned context management as an existing direction. Our focus is an execution process in which context replacement and the creation of agent instances are both policy-selected operations, trained alongside ordinary task actions.

Optimisation across changing contexts. Several methods already train policies whose successive outputs have different conditioning histories. MemAct's Dynamic Context Policy Optimization reconstructs training segments around memory edits and applies trajectory-level advantages to those segments (Zhang et al., 2026). MemSearcher's multi-context GRPO likewise propagates task-level credit across turns with separately constructed contexts (Yuan et al., 2026). Context-Folding is especially close: its preprint formulation jointly trains branching, subtask execution, and return summaries under the appropriate folded contexts (Sun et al., 2026). That formulation's plan-execution instantiation uses interleaved, non-nested branches. FoldAct studies optimisation stability and training efficiency through separate summary and action losses, full-history consistency regularisation, and selective segment training (Shao et al., 2025). CompactionRL jointly optimises execution and summary generation across runtime-triggered compactions (Li et al., 2026). On the other hand, our method, CARP, builds on the same requirement to evaluate actions under their actual generation contexts. It expresses this requirement independently of a particular RL estimator and organises recorded contexts into token trees, covering both history replacement and delegated execution. In particular, its compaction timing is chosen by the policy, whereas CompactionRL triggers compaction when remaining capacity falls below a threshold.

Recursive and parallel agents. Recursive Language Models place long inputs in a persistent programming environment and use model-generated programs to inspect them and invoke recursive calls; the updated study includes both distillation and reinforcement-learning experiments (Zhang et al., 2025). Recursive Agent Optimization jointly trains a shared policy over dynamically generated delegation trees, using node-level rewards that may incorporate root-outcome proxies (Gandhi et al., 2026). Thus, learning delegation and training child agents from task outcomes are also established approaches. CARP formulates delegation together with compaction as operations on a set of threads, without prescribing whether runnable threads execute sequentially or concurrently.

Computation-aware execution. Parallel-Agent Reinforcement Learning (PARL) trains an orchestrator with frozen workers and constrains training and evaluation using critical steps, which account for the longest worker branch at each stage (Kimi Team, 2026). CARP instead defines weighted token-work measures for both the complete execution and its critical path, and combines them in a bounded cost penalty on successful episodes. These measures assume prefix reuse and are computation proxies rather than measured latency. Our experiments demonstrate learned meta-tool use and transfer within CARP.

4 Concurrent Agent Rollout Processes

Context-management mechanisms such as compaction and subagents affect different aspects of agent execution: compaction replaces an agent's interaction history with a condensed history, while spawning a subagent starts another agent instance with a separate interaction history. We introduce the Concurrent Agent Rollout Process (CARP) to describe these mechanisms within a common execution model. CARP makes the state of agent execution explicit, allowing these mechanisms to be expressed as operations on that state.

The state of CARP is a set of threads W. Each *thread* represents an agent instance and maintains its message history and execution status. A thread is *runnable* when it is ready for its next model call, *blocked* when it is waiting for a result, and *terminated* when its execution has ended. Algorithm 1

specifies how CARP advances runnable threads without prescribing their scheduling or requiring parallel execution[1].

Algorithm 1 Concurrent Agent Rollout Process (CARP)

(a) Process execution

Require: initial chat history c, language model p_θ
$W \leftarrow \{\textsc{InitThread}(c)\}$
while W contains a runnable thread **do**
  $w \leftarrow$ any runnable thread in W
  $m \leftarrow \textsc{ThreadStep}(w, p_\theta)$
  for $a \in m.\textsc{tool_calls}$ **do**
    $\textsc{ExecuteTool}(a; w, W)$
  end for
end while

(b) Thread step

Require: thread w, language model p_θ
$\mathbf{x} \leftarrow \textsc{Encode}(w.\textsc{history})$
$\mathbf{y} \sim p_\theta(\cdot \mid \mathbf{x})$
$m \leftarrow \textsc{Decode}(\mathbf{y})$
append m to $w.\textsc{history}$
return m

CARP exposes its own process state through the same tool interface used to interact with the external environment. We call tools that modify this state *meta-tools*: they can modify individual threads or the set of threads W. In Algorithm 1, EXECUTETOOL applies these state changes as part of executing a tool call.

In this paper, we consider two meta-tools:

- *Delegate* creates a new runnable thread w', initialized with a specified chat history, and adds it to W,
$$W \leftarrow W \cup \{w'\}.$$
The calling thread w is blocked until w' terminates, at which point the subagent's result is appended to w's history and w becomes runnable again.
- *Compaction* replaces the calling thread's chat history h with a compacted history h',
$$w.\textsc{history} \leftarrow h',$$
after which execution of w continues normally.

Training with non-linear histories. Meta-tools can replace a thread's history or create another thread with a separate history. Retaining each call's input–output pair is sufficient to evaluate the generated-token log-probabilities. Suppose a realized execution contains J model calls, where call j receives encoded context $\mathbf{x}^{(j)}$ and samples

$$\mathbf{y}^{(j)} = (y_1^{(j)}, \ldots, y_{n_j}^{(j)}) \sim p_\theta(\cdot \mid \mathbf{x}^{(j)}).$$

The model-generated contribution to the execution likelihood is the product of the conditional probabilities at these calls:

$$\prod_{j=1}^{J} p_\theta\left(\mathbf{y}^{(j)} \mid \mathbf{x}^{(j)}\right).$$

Its log-likelihood decomposes over model calls and tokens as

$$\sum_{j=1}^{J} \log p_\theta\left(\mathbf{y}^{(j)} \mid \mathbf{x}^{(j)}\right) = \sum_{j=1}^{J} \sum_{i=1}^{n_j} \log p_\theta\left(y_i^{(j)} \mid \mathbf{x}^{(j)}, y_{<i}^{(j)}\right).$$

The recorded sequences may share prefixes, allowing their token log-probabilities to be computed once and reused. To expose this shared structure, we incrementally build a token tree, a prefix tree in which each non-root node represents one token, by recording each model call's concatenated input–output sequence during CARP execution. Let V be its set of token-bearing nodes and x_v the token at node $v \in V$. Let $\mathrm{Anc}(v)$ denote the ordered sequence of tokens on the path from the root to v,

[1]This applies to tool calls as well. For instance, a thread could block while a tool call is made or work asynchronously without changing the key method.

excluding v, and let n_v count how often x_v was sampled at that prefix. The same log-probability sum becomes

$$\sum_{v \in V} n_v \, \log p_\theta(x_v \mid \mathrm{Anc}(v)) \, .$$

Reusing tokens as context does not increase n_v; context-only nodes have $n_v = 0$. The terms can be evaluated using tree attention or separate root-to-leaf sequences, counting each sampling occurrence once.

The log-probability terms in this sum can be differentiated and used in the gradient-based optimisation methods discussed in Section 2.

Execution topology does not constrain optimization. The decomposition above separates the structure of an agent's execution from the computation required to optimize its policy. Regardless of how threads are created, resumed, or compacted, training requires only the policy-generated tokens and the contexts under which they were sampled. Moreover, the corresponding log-probabilities can be evaluated separately and their gradients accumulated. The total token budget of an execution therefore need not fit in memory at once: peak training memory depends on the largest individual model evaluation, rather than the total number of tokens generated across the execution. In principle, this permits training on arbitrarily long agent executions, with longer rollouts increasing compute rather than peak accelerator memory.

4.1 Application to Reinforcement Learning

The preceding section establishes how to sample CARP executions and compute their generated-token log-probabilities. To use these quantities in reinforcement learning, we additionally specify a reward function. The reward may depend on the final environment state, the agent's final response, or the full interaction history. A chosen RL method then uses the reward together with the generated-token log-probabilities to construct its training objective.

Example: GRPO. Our experiments illustrate how to use GRPO (Shao et al., 2024) with CARP:

- **Sample and score.** For each task, sample a group of token trees produced by CARP. A task-specific reward scores each tree, for example by checking the agent's final response or resulting environment state.
- **Compute advantages and update.** For each tree, subtract the group's mean reward from its reward to obtain an advantage. Apply this advantage to every sampled-token occurrence in the tree, using the token log-probabilities described above to compute the GRPO objective and update the model.

Computation-aware rewards. The reward can also distinguish successful executions by their computation cost. We define a token-based measure of input processing and generation across all threads, assuming ideal prefix-cache reuse. It can weight both total computation and computation along the longest sequence of dependent model calls. This is a cost model rather than measured runtime. Our experiments compare task-only rewards with a bounded cost penalty applied to successful executions; failures receive a fixed reward regardless of cost. Appendix E gives the cost definitions, reward formula, and parameter settings.

5 Experiments

Our experiments test whether GRPO (Shao et al., 2024) with CARP implicitly teaches agents to use meta-tools effectively. We design the tasks and tool access so that completing them requires context management, i.e. the agent must analyse data whose tokenized length exceeds its context budget, and cannot complete the required analysis within that budget without using meta-tools.

5.1 Experimental Setup

We train Qwen3.8-27B Qwen Team (2026) with thinking enabled. To study context management, we restrict the model's context window to 4,096 tokens. A thread whose context exceeds this limit

terminates with a *context-overflow* error. The agent accesses task files through a Read tool, with Compact and Delegate meta-tools available throughout the main experiments (Appendix B). Making these meta-tools available does not ensure that the initial policy uses them before exhausting its context, motivating the interventions below.

5.2 Enabling Context-Management Learning

For reinforcement learning to improve context management, the initial policy must be able to reason about its remaining context capacity and assign non-zero probability to using meta-tools in response. Without sampled meta-tool use, training provides no direct learning signal about its consequences. We observed obstacles to both requirements and address them below. Details and supporting experiments are presented in Appendix D.

Context awareness. We observed that models could not reliably state how many tokens were in their context. Without this information, the agent cannot reliably judge how much context remains before deciding whether to continue or invoke a meta-tool. We therefore modify the tokenizer to supply counters for current context usage and the size of each tool result, giving the agent an explicit signal of its remaining capacity.

Eliciting meta-tool use. Even with explicit token counts, the initial policy demonstrated zero use of meta-tools to avoid context overflow. This left RL with few examples from which to learn effective context management. We therefore provide a coached system prompt explaining the counters and context limit, asking the agent to consider its remaining capacity, and suggesting delegation for self-contained subtasks. It supplies no counting procedure or fixed tool sequence. All main runs use both interventions.

5.3 Learning on Table Counting

We first train the agent on a synthetic context-management task: we construct tables and ask it to count rows matching a specified column value. For example: the agent is asked to count how many rows in the table have column `day=sunday`. The agent runs in a sandbox containing a single file with one table row per line, which it accesses through a Read tool. Each file's tokenized length exceeds the agent's context budget, allowing us to study whether training teaches effective use of meta-tools on a straightforward counting task. The agent succeeds if it returns the correct count. Generation and evaluation details are in Appendices A and C.

We train for up to ten steps, using two seeds each for task-only and cost-sensitive rewards. Each checkpoint is evaluated on 128 held-out episodes; full settings are in Appendix B.

Training results. We train three models using an outcome-only reward and two cost-sensitive rewards, with $\alpha = 1$ and $\alpha = 0.5$. Results are shown in Figure 2. All three learn to use meta-tools successfully, but develop different strategies. With outcome-only rewards, the agent learns to compact frequently, since doing so carries no computation penalty. With $\alpha = 1$, compaction and delegation have comparable computation costs, and the agent continues to favour compaction. This preference may reflect compaction being easier to learn initially.

Parallel task decomposition emerges only in the $\alpha = 0.5$ condition, where the reward accounts for reductions in critical-path cost from parallel execution. This model also receives a prompt encouraging parallelism, as detailed in the appendix E. Under this combination, the agent learns to divide the task into subtasks that subagents can execute in parallel.

5.4 Transfer to OOLONG

We next test whether table-counting training helps on OOLONG-synth (Bertsch et al., 2025), which asks questions about collections of texts, such as how many messages are spam. This requires identifying spam from the message text, rather than counting an explicitly supplied table value. OOLONG supplies the data in the prompt; we instead place the data records in a read-only file accessed through Read, retaining the question and task instructions in the prompt. We use the same context budget and meta-tools as for table counting.

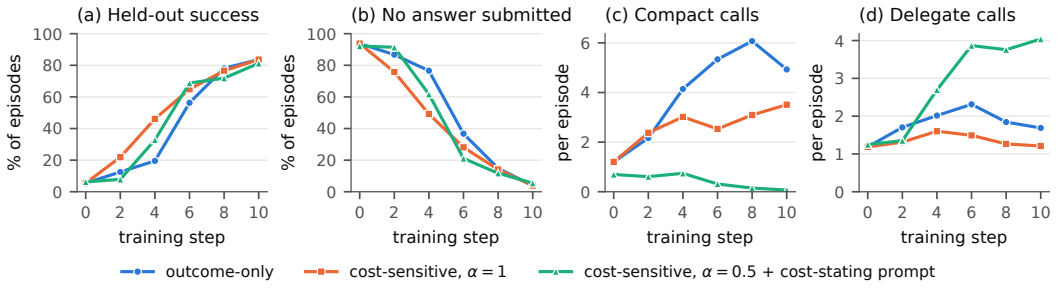

Figure 1: Training on table counting under three reward conditions, one run each. (a) Held-out success. (b) Episodes that end without an answer: context overflow, generation cut-off, timeout or other error. (c, d) Accepted Compact and Delegate calls per episode.

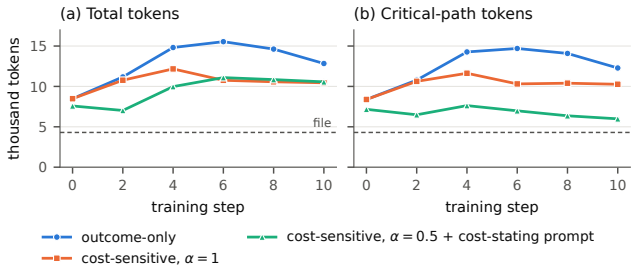

Figure 2: Tokens per held-out table-counting episode during training averaged over all 128 episodes at each checkpoint. Tokens are counted as the cost-sensitive reward counts them: every token read or generated in every thread, with a prefix reused across turns counted once. (a) Total tokens across all threads. (b) Critical-path tokens, which take only the slowest of subagents that ran at the same time. The dashed line is the file itself (4,301 tokens).

Transfer without further training. Without OOLONG training, the table-counting policy raises overall success from 5.7% to 16.2% (Table 1). On questions whose documents exceed the context budget, success rises from 0% to 8.7%.

Further training on OOLONG. We train both initialisations on the same OOLONG questions for ten steps. On questions whose documents exceed the context budget, training from the table-counting policy reaches 34.3% success, compared with 0% when training from the base model (difference: 34.3 percentage points; 95% CI: 26.5–42.5). This supports table counting as an initialisation for learning the new task. The comparison uses one complete run per initialisation, and the table-counting-initialised run includes an optimiser restart.

Longer documents. Performance remains limited at greater lengths. In a separate sweep, success falls to 7.8% at eight times the context budget and to zero at 32 times the budget.

6 Conclusion

We introduced CARP to bring context-management mechanisms such as compaction and subagents into a common framework for agent execution and training. These mechanisms become tools the agent can learn to use through task rewards. By retaining the conditioning context of each model-generated token, we can compute execution log-probabilities and use them in gradient-based training that does not require the complete execution to fit within a single context window. Our experiments demonstrate end-to-end learning of meta-tool use through reinforcement learning from task rewards, and transfer from synthetic table counting to OOLONG. Table-counting training provides an effective initialisation for further OOLONG training: the resulting policy solves tasks whose data exceed the context budget, whereas training from the base model yields no successful executions on those tasks.

Table 1: OOLONG success (%) by document length under a fixed 4,096-token budget, on 128 held-out questions from the OOLONG-synth test split, 4 samples each; n is the number of questions. OOLONG training is ten steps on the same training questions for both initialisations.

			No OOLONG training		After OOLONG training	
Tokens	$\times$ budget	n	Base	Table counting	Base	Table counting
1,024	0.25	6	37.5	54.2	37.5	58.3
2,048	0.5	19	25.0	36.8	32.9	52.6
4,096	1	20	1.3	16.3	0.0	63.7
8,192	2	19	0.0	9.2	0.0	35.5
16,384	4	64	0.0	8.6	0.0	34.0
All		128	5.7	16.2	6.6	42.8

Limitations Learning general context management remains a challenge. Policies may neglect context-capacity signals or adopt task-specific heuristics that transfer poorly. For example, when agents are trained on file analysis tasks where task size does not vary, agents can learn sequences of meta-tool calls without meta-reasoning about it's context capacity. We suspect training across a more diverse set of tasks is required to mitigate against learning task specific meta-tool strategies that do not employ context awareness; establishing this remains future work.

This work lays the groundwork for training agents on tasks whose execution extends beyond a fixed context budget. Potential applications include more context-efficient agents and end-to-end training on longer-horizon tasks.

AI USE STATEMENT

In this work, we used generative AI tools for assisting in implementation of methods and supporting some data analysis, i.e. figure generation. All AI-generated code and analysis was carefully verified and tested by the authors.

We have not used generative AI tools for: generating synthetic data sets, help in development of theoretical models or conceptual frameworks, formulating mathematical claims, proposing or refining hypotheses, designing or providing feedback on research methodology or experiments, cleaning and reformatting dataset, or interpreting results; and providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, propose or refine hypotheses, assist with translation are not applicable to this work.

We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

This work has exclusively used open source models and code. All code associated with the paper will be made publicly available.

REFERENCES

Amanda Bertsch, Adithya Pratapa, Teruko Mitamura, Graham Neubig, and Matthew R Gormley. Oolong: Evaluating long context reasoning and aggregation capabilities. *arXiv preprint arXiv:2511.02817*, 2025.

Apurva Gandhi, Satyaki Chakraborty, Xiangjun Wang, Aviral Kumar, and Graham Neubig. Recursive agent optimization, 2026. URL https://arxiv.org/abs/2605.06639.

Kimi Team. Kimi K2.5: Visual agentic intelligence, 2026. URL https://arxiv.org/abs/2602.02276. Version 2, revised August 2026.

Yujiang Li, Zhenyu Hou, Yi Jing, Jie Tang, and Yuxiao Dong. CompactionRL: Reinforcement learning with context compaction for long-horizon agents. *arXiv preprint arXiv:2607.05378*, 2026. URL https://arxiv.org/abs/2607.05378.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. URL https://arxiv.org/abs/2310.08560.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026. URL https://qwen.ai/blog?id=qwen3.8.

Jiaqi Shao, Yufeng Miao, Wei Zhang, and Bing Luo. FoldAct: Efficient and stable context folding for long-horizon search agents, 2025. URL https://arxiv.org/abs/2512.22733.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon agent via context folding. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=lNRgWoGfYg. Method details discussed here follow the arXiv v1 formulation.

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*, 2023. URL https://arxiv.org/abs/2210.03629.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Jingren Zhou, Siheng Chen, and Yong Jiang. AgentFold: Long-horizon web agents with proactive context folding. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/fa948624dfde013671e72c1a7ca4aebc-Abstract-Conference.html.

Qianhao Yuan, Jie Lou, Zichao Li, Jiawei Chen, Yaojie Lu, Hongyu Lin, Le Sun, Debing Zhang, and Xianpei Han. MemSearcher: Iterative memory integration for search agent via end-to-end reinforcement learning. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 14965–14977, San Diego, California, United States, July 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.findings-acl.736. URL https://aclanthology.org/2026.findings-acl.736/.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2025. URL https://arxiv.org/abs/2512.24601v3. Version 3, revised 11 May 2026.

Yuxiang Zhang, Jiangming Shu, Ye Ma, Xueyuan Lin, Shangxi Wu, and Jitao Sang. Memory as action: Autonomous context curation for long-horizon agentic tasks. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 19149–19164, San Diego, California, United States, July 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.findings-acl.956. URL https://aclanthology.org/2026.findings-acl.956/.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Bryan Kian Hsiang Low, and Paul Pu Liang. MEM1: Learning to synergize memory and reasoning for efficient long-horizon agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=XY8AaxDSLb.

A Counting Task: Generation and Evaluation

Each row is rendered as space-separated `key=value` fields in `data.txt`. Three queryable columns are drawn from day, direction, animal, material, shape, colour, and number. A fourth, `payload`, contains random alphanumeric text and is never queried. The generator calibrates the payload to the desired file token count, independently of row count. The task instruction reveals neither the row count nor the file size. For example, a question is: "How many rows in data.txt have the value 7 in the 'number' column? Return only the count as an integer."

The reported main sets contain 40–120 rows per table, with answers spanning 5–90. Answers are drawn without repetition until the answer pool is exhausted. Matches are distributed using per-table random weights.

Success and coverage. The answer parser strips surrounding whitespace and accepts only a string consisting of digits whose integer value equals the target. A normal stop requires an assistant final message without tool calls and no thread left in an error or timeout state. The primary success metric does not require a particular meta-tool. Coverage is audited separately from numbered Read outputs across all threads. A stricter diagnostic additionally requires a real compaction, rows observed before that reset, previously unseen rows observed afterwards, and full coverage. This diagnostic is specific to compaction and is not the headline metric for policies that may delegate.

B Training and Execution Details

Training and evaluation protocol. Training uses GRPO (Shao et al., 2024) with 32 samples for each of 6 questions per step (192 episodes), group-centred advantages without standard-deviation normalisation, and the episode advantage applied to sampled tokens in every thread. We train for up to 10 steps and evaluate checkpoints every two steps. Each evaluation comprises four samples on each of 32 held-out tables (128 episodes), disjoint by document hash from the 96 training tables. Sampling temperature is 1.0 in both training and evaluation. We report the full checkpoint series rather than treating the best observed checkpoint as an independently selected test result.

The 4,096-token limit applies to inference context length, the per-turn generation cap, trainer sequence length, and capacity stated in the prompt.

Table 2: Training configuration supplied with the experiment records.

Setting	Value
Model	Qwen3.8-27B, thinking enabled
Context / sequence limit	4,096 tokens
Samples per question	32
Questions per step	6 (192 episodes)
Training steps	Up to 10; checkpoints every 2
Optimiser	AdamW, constant learning rate 3×10^{-6}
Optimiser parameters	$\beta = (0.9, 0.999)$; weight decay 0.01; gradient-norm clip 1.0
Precision	fp32 master weights and optimiser state
Micro-batch token budget	1,024
Sampling temperature	1.0 (training and evaluation)
Training execution limits	40 turns, 900 seconds
Evaluation execution limits	80 turns, 1,200 seconds
Hardware	Two nodes with four GH200 devices each; six trainer ranks and one inference device
Distribution	FSDP2; weight updates transferred through the shared filesystem
Evaluation	Four engines; four samples on each of 32 held-out tables

Optimisation. Episodes are represented using token trees and expanded into per-thread sequences for the hybrid attention/Mamba architecture. Each sampled token receives its episode's group-centred advantage A. The reported DPPO loss for a token is

$$\mathcal{L}_t = -m_t A \exp(\ell_t) + 0.001\ell_t^2,$$

where ℓ_t is the log-ratio relative to the recorded sampling probability. The mask m_t excludes policy-gradient terms whose probability ratio moves more than 0.2 against the advantage's sign. The loss is averaged over sampled tokens in the step.

Tool and context semantics. Read accepts a file path, offset, and limit, returning at most 10 lines and 6,000 bytes, with one-based line numbers. Compact retains the original instruction and replaces subsequent history with the supplied summary; the retention rule is fixed by the runtime. Delegate gives a subagent the system instructions and a self-contained task string, with access to the same file. Only the final subagent message is returned to the parent. Subagents have Read and Compact but cannot delegate recursively. The tokenizer inserts `[ctx:N]` before reasoning and `[tokens:K]` after tool results. Context overflow or generation cut-off ends the attempt without recovery.

C Training Diagnostics

Held-out outcomes across runs. The initial policy succeeds on 7/128 episodes, while context overflow and generation cut-off account for 94% of episodes. These failures fall to 2–4% in the three runs evaluated at step 10. The fourth run has 47% context failures at step 8 and the lowest best-checkpoint success. In the task-only run whose success later declines, wrong answers rise from 24% to 40% and timeouts from 2% to 16%, while context failures account for 4% at the final checkpoint. Across all 19 trained checkpoints, no successful episode omits reading or meta-tool use. This audit establishes their presence; full read coverage is assessed separately below.

Initial performance under context pressure. Table 3 compares the untrained policy across file sizes. Accuracy falls from 71% at one quarter of the window to 5% on the training size. At the largest size, 120 of 128 episodes fail through context overflow or generation cut-off, while only one terminates with an incorrect answer. File size alone understates context pressure: instructions, tool formatting, and accumulated reasoning also occupy the window, so even a half-window file can exhaust the available context during an episode.

Table 3: Untrained table counting performance with the coached prompt and both meta-tools. Each row contains 128 episodes.

File tokens	Window ratio	Correct	Wrong answer	No answer
1,024	0.25×	91 (71%)	17	20
2,048	0.50×	26 (20%)	2	100
4,301	1.05×	7 (5%)	1	120

Training improves execution completion. Episode outcomes help explain the improvement. In the task-only training trajectory analysed, the fraction of episodes that overflow falls from 69% to 15%, while normal termination rises from 3% to 74%. Among normally terminating episodes, mean task reward remains between 0.80 and 0.99. These observations suggest that much of the improvement comes from reaching a correct final answer before exhausting the context, rather than from improving answer accuracy conditional on termination. They describe one training trajectory, rather than an average over the four held-out runs.

Dependence on the coached prompt. To test whether the learned behaviour persists without the training-time coaching, we evaluate checkpoints with the explanatory prompt removed. Under the compaction-specific state-carrying criterion (Appendix C), success rises from 0/128 before training to 14/128 at the final evaluated checkpoint. This provides evidence of some learned context management beyond the prompt alone, but the large drop relative to coached evaluation shows that the policy remains sensitive to how its tools and context limits are presented.

Success requires a normal episode termination and the exact integer answer; an error or timeout in any thread invalidates the attempt. We separately audit read coverage across all threads using the line numbers returned by Read. To check for simple shortcuts, we evaluate constant-answer and first-page extrapolation baselines, which score 3.1% and 0% on the held-out set, respectively. As a supporting diagnostic, we also measure strict state-carrying success: an exact answer with full read coverage, a compaction, and previously unseen rows read after that compaction. This criterion checks one form of context management; ordinary task success remains the primary metric because valid delegation strategies need not compact.

Coverage after training. A common failure mode is for the agent to guess the solution to the task without reading the entire data file. We audit coverage (whether a rollout reads all of the data file) and find no successes without reading; among the conditions audited from step 8 onward, every successful episode reads the whole table. This supports effective task completion through extended execution, while coverage alone does not establish the correctness of intermediate reasoning.

D Prerequisite Diagnostics

D.1 Context Awareness

The tokenizer inserts `[ctx:N]` before reasoning to report current context usage and `[tokens:K]` after tool results to report their token count, since the model's own estimates of its usage are poor (Figure 3).

D.2 Eliciting Meta-Tool Use

Coached prompt. The prompt states the context limit, explains both counters, and warns that exceeding the window ends the attempt. It asks the model to weigh remaining capacity before each action, explains what Compact discards, and suggests Delegate for self-contained work. The ordinary coached prompt does not mention the reward. The additional cost-aware prompt asks for less total work and explains concurrent subagents; it also revises the description of delegation from "costs you almost nothing here" to "takes no room in your context." These wording changes must be considered together when interpreting the parallel-decomposition comparison.

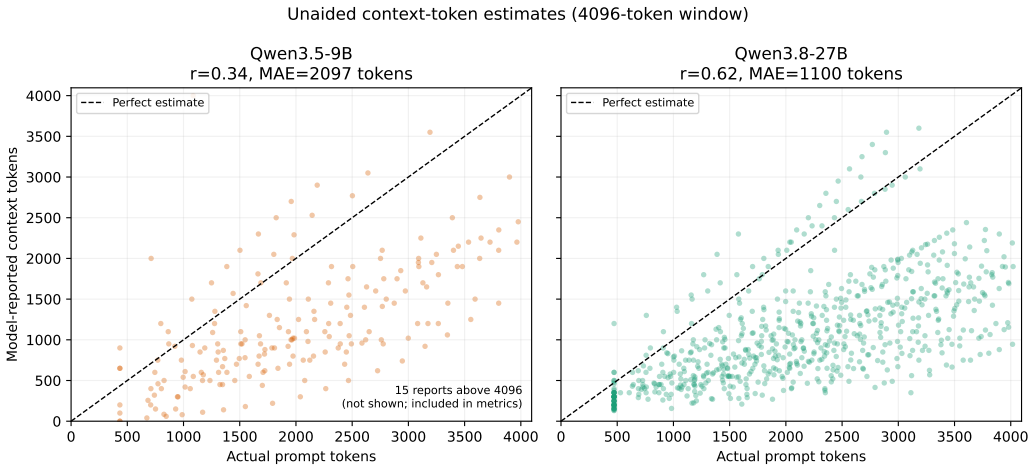

Figure 3: How well the model knows its own context occupancy, unaided. Each point is one assistant turn of an interval-stopping task under a 4,096-token window, run on the unchanged model with its ordinary tokenizer and no count tokens. The task asks the model to begin every reply with `[ctx: N]`, its estimate of the tokens already in its context: the y-axis is that N, and the x-axis is the true count, the length of the exact prompt-token array the model was given for that turn. Turns without a stated count are excluded (6 of 692 for Qwen3.8-27B, 655 of 868 for Qwen3.5-9B). Over the remaining turns we report the Pearson correlation between stated and true counts, the slope of a least-squares fit of stated on true, and the error of the stated count: its mean (1,061 tokens too low for 27B), its mean absolute value (1,100 tokens) and the share of estimates within 10% of the true count (4.7%). A perfect estimate lies on the dashed line; 27B's fitted slope is 0.40. Fifteen of 9B's reports exceed 4,096 and are not drawn but are included in the statistics. 56 documents per model, one attempt each; turns within an episode are correlated, so the points are not independent.

The main runs use the coached prompt and explicit context counters, without prefix SFT. A preliminary prefix-only SFT intervention produced premature stopping: the experiment brief records 104 of 109 completions with no answer. This intervention was abandoned rather than used to initialise the reported policies.

Read size and initial success. With the untrained policy, reported success is 7/128 with 10-line pages, 1/128 with 25-line pages, and 0/192 with 50-line pages. Larger responses can consume the room needed for reasoning and further actions. These observations motivate using small pages to obtain occasional successful trajectories for outcome-based training; they do not establish a universal page-size threshold or prove that training with larger pages is impossible.

E Computation Cost and Reward Definitions

We define a token-based computation-cost measure for a CARP execution containing J model calls, with n_j tokens sampled by call j. The *output cost* is

$$O = \sum_{j=1}^{J} n_j.$$

Contexts, by contrast, overlap: an invocation's context typically extends an earlier context together with that invocation's output, and after compaction it still begins with the retained instructions. Let $\mathcal{P}_j = \{\mathbf{x}^{(j)}_{\le m} \mid 1 \le m \le |\mathbf{x}^{(j)}|\}$ be the prefixes of context j, and let $\mathcal{S}_j = \{\mathbf{x}^{(j)} \parallel y^{(j)}_{\le i} \mid 1 \le i \le n_j\}$ be the prefixes that end in a token the policy sampled itself. A prefix needs to be processed once however many contexts share it, and a sampled token has already been processed when it was

generated. The *input cost* is therefore

$$I = \Big| \bigcup_{j=1}^{J} \mathcal{P}_j \;\setminus\; \bigcup_{j=1}^{J} \mathcal{S}_j \Big|,$$

and the total computation is $C_\Sigma = w_{\text{in}} I + w_{\text{out}} O$.

Parallel subagents. Invocations need not run one after another. Write $j' \prec j$ when invocation j cannot start before j' has finished: consecutive invocations of the same agent; an invocation that delegates and the first invocation of each subagent it starts; and the last invocation of each subagent and the delegating agent's next invocation. Subagents started by the same invocation are unordered and may run concurrently. Let $\prec^*$ be the transitive closure of $\prec$. Invocation j must itself process the input that none of the invocations it depends on had already processed,

$$I_j = \Big| \mathcal{P}_j \setminus \mathcal{S}_j \setminus \bigcup_{j' \prec^* j} \big(\mathcal{P}_{j'} \cup \mathcal{S}_{j'}\big) \Big|,$$

and the *critical-path cost* is the heaviest chain of dependent invocations,

$$C_{\max} = \max_{j_1 \prec j_2 \prec \cdots \prec j_L} \sum_{l=1}^{L} \big(w_{\text{in}} I_{j_l} + w_{\text{out}} n_{j_l}\big).$$

$C_{\max}$ is the computation on the longest sequence of invocations that must happen one after another: the episode's latency with unlimited parallel workers. It never exceeds C_Σ, and equals it when no two invocations are unordered. We price an episode by

$$C_\alpha = \alpha C_\Sigma + (1-\alpha) C_{\max}, \quad \alpha \in [0,1].$$

As with the likelihood, meta-tools change which contexts arise and which invocations depend on which, but not how the cost is evaluated.

RL rewards. We normalise computation by a task-specific reference r and subtract a baseline b, giving $c = C_\alpha / r - b$. For a binary task outcome $s \in \{0,1\}$, the cost-sensitive reward is

$$R = \begin{cases} 1 - \varepsilon\big(1 - e^{-\max(c,0)/\tau}\big) & s = 1, \\ -1 & s = 0. \end{cases}$$

The cost penalty is bounded by ε and applies only to successful attempts. Failed attempts receive the same reward regardless of their cost. With group-centred advantages and no standard-deviation normalisation, variation in successful episodes' costs can provide a learning signal even when every episode in a group succeeds. For table counting we use the file's token count as r, $b = 1$, $\varepsilon = 0.2$, and $\tau = 1.5$. The task-only condition uses the binary reward $R = s$. Consequently, the two conditions also differ in the scale of their success–failure reward gap; this comparison does not isolate cost shaping from reward scale.

F Cost and Behaviour Analyses

Effect of computation cost. We compare task-only and cost-sensitive training with both tools available, using total token-tree cost ($\alpha = 1$) and equal input/output weights. With two seeds per condition, the results do not establish that the cost objective improves accuracy, reduces whole-task hand-offs, or lowers the cost of successful episodes. At step 10, cost-sensitive seed 1 leads task-only seed 1 by 43 percentage points, but task-only seed 2 reaches the same 84% success. Delegate calls per episode overlap across conditions: 1.1–2.3 for task-only and 1.2–1.8 for cost-sensitive training. The success–failure reward gap also differs, at 1 for task-only and 1.8–2.0 for cost-sensitive training, confounding the effect of cost shaping with reward scale. At step 8, cost-sensitive success rates are 77% and 48%, versus 66% and 78% for task-only training. Successful-episode costs also overlap: 2.27 and 3.42 times the file size with cost-sensitive training, versus 2.58 and 2.91 with task-only training (Appendix F). The variation across seeds prevents us from attributing the observed strategy differences to the cost objective.

Table 4: Step-8 comparison across two seeds per reward. Costs are normalised by file size and conditioned on success.

	Task-only		Cost-sensitive	
	Seed 1	Seed 2	Seed 1	Seed 2
Success (%)	66	78	77	48
Whole-task hand-offs / episode	1.93	1.70	0.99	1.55
Cost / successful episode	2.58	2.91	2.27	3.42

The token-tree accounting counts shared prefixes once and all sampled outputs, including subagents. This is a token-based compute proxy under ideal prefix reuse, rather than measured wall-clock latency. The supplied critical-path implementation assigns shared prefix costs incrementally to threads; identical sibling prefixes are assigned to the first sibling. This accounting approximation should be distinguished from execution latency.

F.1 Strategies and Execution Examples

Page-and-compact executions preserve a running total and next unread line in a summary. Whole-task hand-offs instead give a fresh subagent the complete question; that subagent may perform its own compactions. Line-range slicing assigns explicit subsets to subagents, while parallel fan-out starts several such subagents in the same parent turn.

In an exploratory arm combining $\alpha = 0.5$ with the cost-aware prompt, 119/128 step-8 episodes use parallel fan-out, with a median of four subagents per fork and a maximum of nine. The reported critical-path cost per success is 1.30 times file size, versus 2.25 in the comparison arm; correct episodes are 92/128 versus 98/128. An arm with the same α and the ordinary coached prompt produces 8, 6, and 0 parallel episodes at steps 4, 6, and 8, respectively, and solves 47/128 at step 8. These single-run comparisons suggest that prompt wording is important; they do not isolate a causal effect of the cost weighting or of any individual prompt sentence.

Observed failures include context exhaustion, undercounting near hand-offs, and repeated whole-task delegation until timeout. In one final checkpoint, 20/128 episodes reach the 1,200-second limit after a median of five hand-offs, compared with two overall. Because subagents cannot delegate, these repetitions are initiated by the parent.