WHEN TO END, WHICH TO INVOKE NEXT: TRANSITION-AWARE TOKENIZED PROCEDURAL MEMORY FOR LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

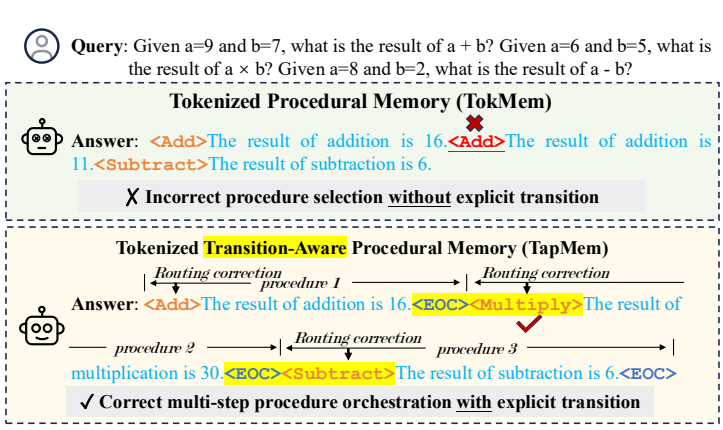
One-Token Procedural Memory (*TokMem*) has recently emerged as a parameter-efficient paradigm for storing and reusing task-solving procedures in large language models (LLMs), where each procedure is encoded as a compact memory token. This paper investigates a more challenging scenario in which *a LLM must reliably orchestrate a sequence of consecutive procedure recalls within a single generation*. We identify a **fundamental limitation** of TokMem-style procedure chaining: while memory tokens provide explicit entry points to individual procedures, the transitions between procedures remain implicit. This overlooked handoff mechanism causes next-procedure selection to become increasingly fragile as procedural chains grow longer. To address this issue, we propose *Transition-Aware Procedural Memory* (*TapMem*), a lightweight framework that explicitly models inter-procedure handoffs. The key insight behind TapMem is to formulate each procedure transition as a boundary-conditioned selection problem. Since the next procedure should be selected only after the current procedure-guided span ends, TapMem first introduces **E**xplicit **T**ermination **S**ignal, to make this completion boundary observable. Building on the resulting boundary representation, TapMem further incorporates a **T**ransition-**C**onditioned **R**outing **A**dapter to generate a context-dependent correction over candidate memory-token logits. By intervening only at transition positions and only over memory-token logits, TapMem preserves the original full-vocabulary autoregressive generation process while improving the sparse procedural decisions required for reliable multi-step orchestration. Extensive experiments across diverse procedure-recall settings demonstrate that TapMem achieves promising performance.

1 INTRODUCTION

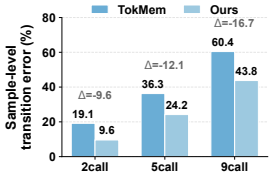
Large language models (LLMs) (Grattafiori et al., 2024; Yang et al., 2025) are increasingly deployed in tasks where success depends not only on generating coherent responses, but also on executing structured task-solving procedures (Fang et al., 2026; Han et al., 2026). These tasks often require models to execute **multiple interdependent procedures**, such as invoking external tools, reasoning over intermediate results, verifying partial outputs, and formatting final responses (Liu et al., 2024a; Zhou et al., 2024; Deng et al., 2023; He et al., 2024). Such behaviors are inherently *procedural*: they specify how the model should act under a given context, rather than what factual content should be retrieved. As LLM-based applications become increasingly structured (Guo et al., 2024; Lu et al., 2025; Patil et al., 2025; Abdelaziz et al., 2024; Cai et al., 2024), a central challenge is not only to represent reusable procedures efficiently, but also to compose them reliably within a single generation.

To enable such capabilities, recent studies have investigated procedural memory, with related discussions provided in Appendix A. Among these approaches, tokenized procedural memory offers a promising paradigm for this goal. In its typical formulation, One-Token Procedural Memory (TokMem) (Wu et al., 2026) encodes each reusable procedure as a compact memory token, thereby casting *procedural recall as a language-modeling problem*, where invoking a procedure amounts to predicting its corresponding memory token. This design enables parameter-efficient invocation of individual procedures, but it does not explicitly model the *transitions* required when multiple procedures must be executed consecutively. In such scenarios, the memory-token prediction mechanism

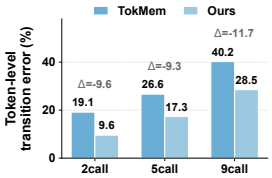
Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping 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.



(a) Toy example of transition-aware procedural memory.



(b) Comparison of transition error in sample level.



(c) Comparison of transition error in token level.

Figure 1: **(a)** A toy example of *TokMem* and *TapMem* (Ours). TokMem incorrectly selects the second procedure, while TapMem enables reliable procedure chaining through explicit transition modeling. **(b)(c)** TapMem achieves a larger reduction in sample-level and token-level transition errors, and this advantage grows as the number of calls increases. More details can be found in Appendix D.2.1.

implicitly serves dual roles: identifying the completion of the current procedure and selecting the next procedure, which may cause compositional reliability to degrade as procedure chains grow longer. Figure 1a provides a toy example of this failure mode, where TokMem selects the first procedure correctly but fails at the subsequent transition. Figure 1b and Figure 1c further show that its transition errors become sharp with longer procedure chains, motivating explicit transition modeling for reliable next-procedure selection. This leads to our central question:

How to develop a more reliable framework for multi-step procedural orchestration?

To answer this question, we propose *Transition-Aware Procedural Memory* (*TapMem*), a lightweight framework that augments tokenized procedural memory with explicit transition modeling. Our key insight is that **reliable multi-step procedural orchestration requires separating two inherently different decisions during a procedure transition**: determining when the current procedure has completed and selecting which procedure should be invoked next. TapMem therefore treats inter-procedure transitions as boundary-conditioned routing problems (Geng et al., 2023; Beurer-Kellner et al., 2023; Ross et al., 2025). Specifically, TapMem introduces an Explicit Termination Signal (ETS) to make procedure completion observable by requiring the model to emit a `<EOC>` token at the end of each procedure-guided span. This explicit boundary representation enables a Transition-Conditioned Routing Adapter (TCRA), which dynamically adjusts candidate memory-token logits based on the transition context for next-procedure selection. By restricting its intervention to transition positions and memory-token logits only, TapMem preserves the original full-vocabulary autoregressive generation process while enhancing the sparse procedural decisions required for reliable multi-step orchestration. Extensive experiments across standard, long-horizon, multi-turn execution, and continual procedural memory settings demonstrate that TapMem improves procedural orchestration over TokMem, with increasingly larger advantages on longer procedure chains.

Our main contributions are summarized as follows:

- We identify the transition bottleneck in tokenized procedural memory, where procedure termination and next-procedure selection are implicitly coupled.
- We propose TapMem, a transition-aware framework that explicitly models procedure boundaries and improves next-procedure routing with ETS and TCRA.
- We evaluate TapMem across diverse procedural memory settings, including long-horizon orchestration, multi-turn execution, and continual extension, demonstrating improved reliability over TokMem and the potential for scalable procedural memory.

2 METHODOLOGY

2.1 PROBLEM FORMULATION & MOTIVATION

This paper studies *tokenized procedural memory* for multi-step procedure orchestration in LLMs. Following prior work on TokMem (Wu et al., 2026), a *procedure* is defined as a reusable context-response or task-solving routine that specifies how a model should act in a given context, such as task decomposition, tool invocation, and response formatting. To enable LLMs to manage and reuse such procedures efficiently, TokMem represents each reusable procedure with a compact memory token, which serves as a learned handle for eliciting the corresponding behavior from the language model. Formally, let f_θ denote a pretrained LLM with the original vocabulary $\mathcal{V}$, where θ denotes the backbone parameters. TokMem first augments the original vocabulary with K special procedural memory tokens:

$$\widetilde{\mathcal{V}} = \mathcal{V} \cup \mathcal{V}_M, \qquad \mathcal{V}_M = \{m_1, m_2, \ldots, m_K\}, \qquad \mathcal{V} \cap \mathcal{V}_M = \emptyset \tag{1}$$

here, $\mathcal{V}_M$ extends the original language-model vocabulary with procedural memory tokens. Each memory token $m_i \in \mathcal{V}_M$ is assigned a new token index in the augmented vocabulary $\widetilde{\mathcal{V}}$ and is associated with a trainable embedding vector $\mathbf{e}_i \in \mathbb{R}^d$, where d is the embedding dimension. The trainable embeddings of all procedural memory tokens form a memory bank:

$$\mathbf{E} = \begin{bmatrix} \mathbf{e}_1^\top \\ \mathbf{e}_2^\top \\ \vdots \\ \mathbf{e}_K^\top \end{bmatrix} \in \mathbb{R}^{K \times d} \tag{2}$$

Given an input query, the goal is to generate a response by invoking **one or more appropriate procedures** indexed by $\mathcal{V}_M$. Suppose that a training instance with $J \geq 1$ procedure calls is serialized as

$$\mathbf{y} = \left[\mathbf{q},\, m^{(1)},\, \mathbf{a}^{(1)},\, m^{(2)},\, \mathbf{a}^{(2)},\, \ldots,\, m^{(J)},\, \mathbf{a}^{(J)}\right] \tag{3}$$

where $\mathbf{q}$ denotes the tokenized query, $m^{(i)} \in \mathcal{V}_M$ is the memory token selected for the i-th procedure call, and $\mathbf{a}^{(i)} = (a_1^{(i)}, \ldots, a_{L_i}^{(i)})$ denotes the ordinary response segment generated under that procedure.

TokMem optimizes the trainable memory bank $\mathbf{E}$ using the standard autoregressive next-token prediction objective:

$$\mathcal{L}_{\mathrm{AR}}(\mathbf{y}; \mathbf{E}) = -\sum_{j=|\mathbf{q}|+1}^{|\mathbf{y}|} \log p_{\theta,\mathbf{E}}\left(y_j \mid y_{<j}\right) \tag{4}$$

where $|\mathbf{q}| + 1$ is the position of the first generated token and j denotes a general token position in the sequence.

Discussions The objective in Eq. 4 provides a compact and parameter-efficient mechanism for procedure invoking. However, *we argue that it leaves a key part of compositional procedure orchestration under-modeled.* From the perspective of sequence serialization, procedural memory token $m^{(j)}$ and ordinary response tokens $\mathbf{a}^{(j)}$ are optimized under the same next-token prediction objective in Eq. 4. After completing the j-th procedure invoking, TokMem directly predicts the next memory token as

$$p_{\theta,\mathbf{E}}\left(m^{(j+1)} \mid \mathbf{q}, m^{(1)}, \mathbf{a}^{(1)}, \ldots, m^{(j)}, \mathbf{a}^{(j)}\right), \qquad j < J \tag{5}$$

This formulation implicitly conflates two structurally distinct decisions: whether the current procedure has completed, and which procedure should be invoked next. Such an implicit transition mechanism may be sufficient for isolated procedure recall, but **becomes fragile when** $J > 1$, where reliable procedure chaining requires accurate procedure completion and transition dependencies across

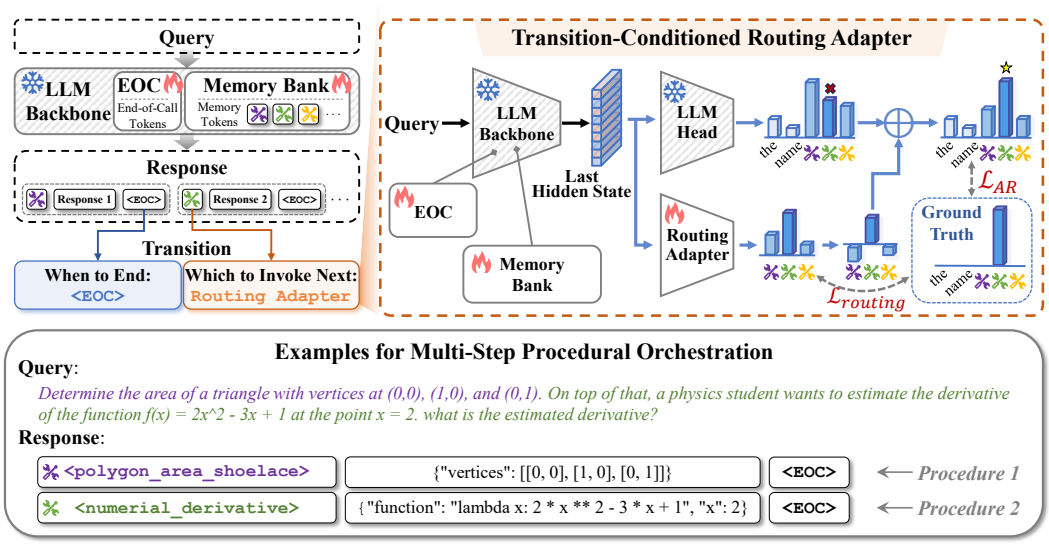


Figure 2: **Overview of our *TapMem* approach.** Given a query, the model selects a memory token from the memory bank to determine the current procedure, then generates the corresponding response, and finally emits <EOC> to mark the end of this procedure. At positions corresponding to memory token generation, TCRA performs routing correction on the LM output logits. The bottom panel shows a case of multi-step procedural orchestration.

multiple procedure calls. To address this challenge, we therefore reformulate tokenized procedural memory as a *transition-aware procedure orchestration* problem, where procedure termination and next-procedure routing are explicitly modeled rather than left to generic autoregressive prediction.

2.2 TapMem: Transition-Aware Procedural Memory

To address the transition bottleneck identified in Section 2.1, we propose *Transition-Aware Procedural Memory* (*TapMem*). TapMem is built on a simple structural principle: the next procedure should only be selected after the completion of the current procedure-guided span. Instead of requiring a single memory-token prediction to implicitly determine both procedure termination and next-procedure selection, TapMem explicitly separates these two decisions. As shown in Figure 2, TapMem realizes this principle with two components: *Explicit Termination Signal* (ETS), and *Transition-Conditioned Routing Adapter* (TCRA), which refines candidate memory-token logits based on the exposed transition state. This design preserves standard autoregressive response generation within each procedure while strengthening the sparse transition decisions required for reliable multi-step orchestration.

2.2.1 When to End: Explicit Termination Signals

The first question in transition-aware procedural memory is **when the current procedure should end**. In TokMem-style chaining, procedure completion is not explicitly modeled and is inferred only from the next memory token, which implicitly serves two roles: terminating the current procedure and selecting the next one. This coupling leaves procedure boundaries under-specified, especially in multi-step procedural chaining.

To expose this boundary, TapMem introduces an Explicit Termination Signal (ETS) implemented as a reserved end-of-call token <EOC>. By inserting this signal after each procedure-guided span, TapMem transforms procedure completion into an explicit autoregressive prediction target, allowing the model to identify procedure boundaries directly rather than infer them from the subsequent memory token. Unlike ordinary textual delimiters or <EOS>, ETS marks the completion of the current procedure without terminating the overall generation sequence. Thus, ETS separates procedure-level completion from sequence-level termination while remaining compatible with standard autoregressive decoding. For convenience, we continue to use $\widetilde{\mathcal{V}}$ to denote the augmented vocabulary containing the termination

signal <EOC>. Formally, a single procedure call is serialized as

$$m, \mathbf{a}, \texttt{<EOC>}, \tag{6}$$

where $m \in \mathcal{V}_M$ is the memory token that selects the current procedure, and $\mathbf{a}$ denotes the response span generated under that procedure. For a multi-step trajectory with J procedure calls, TapMem constructs the target sequence as

$$\mathbf{y}' = \big[\mathbf{q},\, m^{(1)}, \mathbf{a}^{(1)}, \texttt{<EOC>},\, \ldots,\, m^{(J)}, \mathbf{a}^{(J)}, \texttt{<EOC>}\big], \tag{7}$$

where $\mathbf{q}$ is the input query, $m^{(j)}$ is the memory token for the j-th procedure, and $\mathbf{a}^{(j)}$ is the corresponding procedure-guided response span. ETS makes the completion of procedure observable through the generation of <EOC>. We define the set of inter-procedure boundary positions as

$$\mathcal{B} = \big\{ b \in \{|\mathbf{q}|+1, \ldots, |\mathbf{y}'|-1\} \,\big|\, y'_b = \texttt{<EOC>},\ y'_{b+1} \in \mathcal{V}_M \big\}. \tag{8}$$

$b \in \mathcal{B}$ is therefore the position of an <EOC> token that terminates the current procedure and is followed by the memory token of the next procedure.

2.2.2 Which to Invoke Next: Transition-Conditioned Routing Adapter

After ETS explicitly marks the end of the current procedure, the next question is ***which procedure should be invoked next***. Although the original LM head can predict the next memory token, we argue that procedure selection differs structurally from ordinary response generation. At a procedure-selection position, the model should distinguish among candidate procedural memories, while natural-language and argument generation remain governed by the base LM. We therefore introduce the *Transition-Conditioned Routing Adapter* (TCRA) to strengthen memory-token selection without disturbing ordinary generation.

The initial procedure selection occurs at position $|\mathbf{q}|+1$, whereas each subsequent procedure selection occurs after a position in $\mathcal{B}$. Accordingly, we define the set of procedure-selection positions at which TCRA applies routing refinement as

$$\mathcal{R} = \{|\mathbf{q}|+1\} \cup \{b+1 \mid b \in \mathcal{B}\}. \tag{9}$$

where $r = |\mathbf{q}|+1$ denotes the position of the initial procedural memory token, and each $r = b+1$ for $b \in \mathcal{B}$ denotes the position of the next procedural memory token following an <EOC> token. By construction, $y'_r \in \mathcal{V}_M$ for every $r \in \mathcal{R}$.

As shown in Fig. 2, TCRA provides a routing adaptation at procedure-selection positions, where the model needs to choose among candidate procedure memories rather than continue ordinary response generation. Let $\mathbf{h}_{r-1} \in \mathbb{R}^d$ denote the hidden state used to predict the procedural memory token at position $r \in \mathcal{R}$. TCRA maps this boundary representation to routing logits over the K candidate procedural memories:

$$\mathbf{z}_r^{\mathrm{TCRA}} = g_\phi(\mathbf{h}_{r-1}) \in \mathbb{R}^K, \qquad r \in \mathcal{R} \tag{10}$$

where g_ϕ is a lightweight projection adapter and K is the number of memory tokens. The logits $\mathbf{z}_r^{\mathrm{TCRA}}$ encode a transition-conditioned preference over candidate procedure memories, which is later injected only into memory-token logits, so ordinary vocabulary logits remain governed by the base LM head.

We then convert this routing preference into a logits adjustment relative to a uniform reference distribution. Let $\boldsymbol{\pi}_r = \mathrm{softmax}(\mathbf{z}_r^{\mathrm{TCRA}})$, where $\pi_r(k)$ denotes the probability assigned to memory token m_k. For each $m_k \in \mathcal{V}_M$, we define the adjustment as

$$\Delta_r(k) = \log \pi_r(k) - \log(1/K), \qquad r \in \mathcal{R}, \quad k \in \{1, \ldots, K\} \tag{11}$$

where $1/K$ is the probability assigned to each candidate under the uniform distribution. If $\pi_r(k) = 1/K$ for every $k \in \{1, \ldots, K\}$, then $\Delta_r(k) = 0$ for every candidate memory token.

Taking a step further, let $\mathbf{z}_r^{\mathrm{LM}} \in \mathbb{R}^{|\widetilde{\mathcal{V}}|}$ denote the base logits over the augmented vocabulary used to predict y'_r. At a procedure-selection position $r \in \mathcal{R}$, TapMem applies the adjustment only to procedural memory-token logits:

$$\widetilde{z}_{r,m} = \begin{cases} z^{\mathrm{LM}}_{r,m} + \alpha \Delta_r(k), & m = m_k,\ m_k \in \mathcal{V}_M, \\ z^{\mathrm{LM}}_{r,m}, & m \notin \mathcal{V}_M, \end{cases} \qquad r \in \mathcal{R} \tag{12}$$

where $\alpha \geq 0$ controls the strength of the TCRA adjustment relative to the base logits. For positions $r \notin \mathcal{R}$, TapMem directly predicts without adjustment. Therefore, TCRA strengthens procedure selection only at procedure-selection positions, while preserving ordinary full-vocabulary generation inside each procedure-guided span.

2.3 Training and Inference with TapMem

TapMem follows the autoregressive objective $\mathcal{L}_{\mathrm{AR}}$ defined above, but uses adjusted logits at procedure-selection positions. Specifically, for $r \in \mathcal{R}$, TCRA adjusts only candidate memory-token logits according to Eq. 12; at all other positions, the original logits are used unchanged.

Since procedure selection occurs only at sparse positions, $\mathcal{L}_{\mathrm{AR}}$ provides only indirect supervision for the routing adapter. We therefore add a supervised routing loss over the procedure-selection positions. For each $r \in \mathcal{R}$, let $k_r \in \{1, \ldots, K\}$ denote the target procedure index satisfying $y'_r = m_{k_r}$. The routing loss is

$$\mathcal{L}_{\mathrm{route}} = -\frac{1}{|\mathcal{R}|} \sum_{r \in \mathcal{R}} \log \pi_r(k_r) \tag{13}$$

Accordingly, the full training objective is

$$\mathcal{L} = \mathcal{L}_{\mathrm{AR}} + \gamma \mathcal{L}_{\mathrm{route}} \tag{14}$$

where $\gamma \geq 0$ controls the strength of the routing supervision.

During training, the backbone LLM remains frozen, and we update only the added memory-token embeddings, the <EOC> embedding, and the routing adapter g_ϕ. Both $\mathcal{L}_{\mathrm{AR}}$ and $\mathcal{L}_{\mathrm{route}}$ optimize g_ϕ, while gradients from TCRA are not propagated to its input hidden states. The memory-token embeddings and <EOC> embeddings remain trained through the autoregressive objective. This design lets $\mathcal{L}_{\mathrm{route}}$ train the adapter to improve next-procedure selection, without directly reshaping hidden representations used for ordinary response generation. We demonstrate the effectiveness of this design in Figure 8.

During inference, the adapter is employed at procedure boundaries, and the correction it produces is added to the memory-token logits before the next token is selected. Detailed algorithms are provided in Appendix C.

3 Experiments

Due to space limitations, please refer to the Appendix D and E for a extended version.

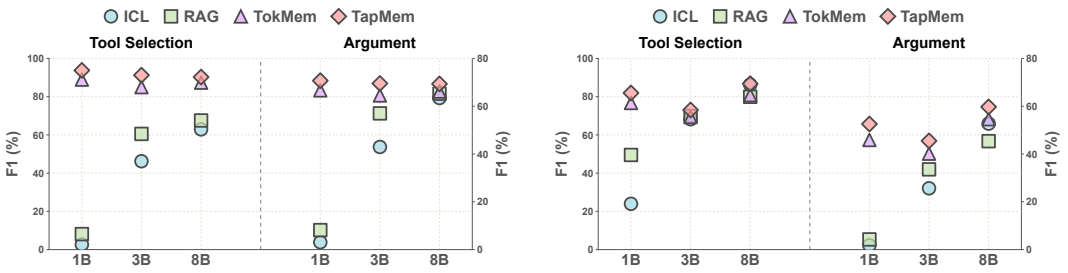
3.1 Experiments Setup

For a comprehensive evaluation, we organize our experiments into **four progressive settings**:

- **Standard setting.** Following TokMem (Wu et al., 2026), we evaluate *compositional recall* on APIGen (Liu et al., 2024b), which requires multiple procedure calls within a single generation, and *atomic recall* on SNI (Wang et al., 2022), which requires a single procedure call.
- **Long-horizon setting.** To evaluate longer generations involving more procedure calls, we conduct experiments on TaskBench (Shen et al., 2024) and TRAJECT-Bench (He et al., 2026), and extend APIGen to sequences of up to 10 procedure calls.
- **Multi-turn execution setting.** To evaluate procedure execution across multiple interaction steps with environmental feedback, we conduct experiments on InterCode-Bash (Yang et al., 2023).
- **Continual setting.** To evaluate the incorporation of unseen procedures, we consider a continual setting with unseen procedures and explore several cold-start approaches.

Table 1: Compositional recall results on the **APIGen** dataset in terms of F1. † denotes training-free methods.

Method	Tool Selection				Argument			
	2 calls	3 calls	4 calls	Avg.	2 calls	3 calls	4 calls	Avg.
Llama 3.2 1B								
ICL†	2.0	1.0	0.0	1.4	1.4	0.4	0.0	0.9
RAG†	15.1	3.8	0.0	8.9	9.7	2.1	0.0	5.5
TokMem	83.2(±1.4)	82.2(±1.2)	76.4(±0.6)	82.4(±1.2)	67.6(±0.5)	67.6(±1.4)	55.2(±3.0)	67.0(±0.7)
TapMem	88.8(±0.9)	89.6(±0.7)	81.0(±2.0)	**88.8**(±0.8)	74.9(±0.1)	74.1(±0.8)	65.0(±2.0)	**74.0**(±0.3)
Fine-Tuning	84.2(±2.2)	80.8(±1.4)	73.7(±3.6)	82.0(±1.7)	79.3(±0.6)	78.2(±0.2)	64.7(±6.2)	78.1(±0.6)
TokMem + adaptation	98.2(±0.5)	98.3(±0.1)	96.1(±1.2)	98.1(±0.3)	84.3(±0.9)	84.1(±0.4)	74.8(±0.8)	83.7(±0.6)
TapMem + adaptation	98.4(±0.4)	98.4(±0.2)	97.2(±0.1)	**98.3**(±0.2)	85.8(±0.9)	85.3(±0.9)	79.0(±1.8)	**85.3**(±0.7)
Llama 3.2 3B								
ICL†	59.4	61.8	56.9	60.4	48.4	49.8	44.1	48.9
RAG†	80.4	77.1	83.6	79.0	55.1	53.1	48.5	53.8
TokMem	73.6(±0.3)	75.3(±1.3)	70.3(±3.6)	74.2(±0.7)	59.7(±1.6)	59.7(±4.1)	52.1(±8.0)	59.3(±3.0)
TapMem	81.1(±0.8)	82.7(±1.6)	81.4(±0.9)	**81.9**(±1.0)	65.5(±3.0)	65.4(±3.7)	58.3(±3.1)	**65.1**(±3.1)
Fine-Tuning	97.7(±0.5)	98.2(±0.5)	94.4(±2.1)	97.8(±0.4)	87.2(±0.7)	87.7(±0.5)	79.2(±2.1)	87.0(±0.2)
TokMem + adaptation	98.2(±0.6)	98.7(±0.2)	97.5(±1.4)	98.4(±0.3)	86.6(±0.7)	86.9(±0.5)	79.4(±0.8)	86.4(±0.5)
TapMem + adaptation	98.9(±0.7)	99.4(±0.3)	98.6(±0.2)	**99.2**(±0.5)	88.0(±0.9)	88.6(±0.7)	82.5(±1.7)	**88.0**(±0.7)
Llama 3.1 8B								
ICL†	57.7	63.6	52.3	60.3	49.6	55.2	49.7	52.3
RAG†	79.0	78.2	74.1	78.4	60.1	60.7	55.1	60.1
TokMem	84.9(±1.2)	83.8(±1.3)	74.3(±5.2)	83.9(±0.2)	71.2(±1.5)	69.7(±1.2)	62.6(±5.1)	70.1(±1.0)
TapMem	87.1(±0.7)	86.5(±0.5)	79.9(±2.2)	**86.4**(±0.4)	74.7(±1.6)	74.3(±1.1)	65.1(±3.3)	**74.0**(±0.7)
Fine-Tuning	92.7(±0.9)	94.4(±0.9)	89.3(±3.3)	93.3(±0.9)	83.2(±1.6)	84.2(±0.4)	78.4(±1.4)	83.5(±0.8)
TokMem + adaptation	97.8(±0.8)	98.1(±0.1)	97.9(±1.8)	97.9(±0.4)	84.9(±2.2)	84.1(±0.9)	82.2(±4.0)	84.4(±1.6)
TapMem + adaptation	98.1(±0.7)	98.1(±0.5)	96.6(±2.1)	**98.0**(±0.6)	86.3(±1.3)	85.5(±0.9)	81.1(±3.7)	**85.7**(±0.9)



(a) Results on TaskBench.

(b) Results on TRAJECT-Bench.

Figure 3: Compositional recall results on the **TaskBench** and **TRAJECT-Bench** in terms of F1.

For the standard, long-horizon and continual settings, we report *Tool Selection F1* and *Argument Prediction F1* to measure procedural memory selection and full function-call correctness, respectively. For the multi-turn execution setting, we report the *success rate* and *reward*. Results are mainly averaged over 3 random seeds, with TokMem (Wu et al., 2026) as the baseline. The TCRA adapter is implemented as a linear projection layer. Detailed introductions are deferred to Appendix D.

3.2 Overall Performance

Results on Standard Setting. (1) *Compositional Recall.* Table 1 reports the results on APIGen. While TokMem provides a stronger baseline through tokenized procedural memory, its performance still degrades as the number of procedure calls increases. In contrast, our TapMem achieves better performance in both tool selection and argument generation, and shows a smaller drop as the number of calls increases, yielding average gains of 6.4% and 7.0%, respectively, on Llama 3.2 1B. We further observe average gains over TokMem of 7.4%/13.2% in Tool Selection F1/Argument Generation F1 on Qwen 3.5 4B, and 3.7%/9.5% on Qwen 3.5 9B (Team, 2026); detailed results are reported in Appendix E.1.1, demonstrating generalization across model families. TapMem also remains effective after adaptation, where the backbone is first fine-tuned on compositional recall data. (2) *Atomic Recall.* Table 3 reports the results on SNI (Wang et al., 2022), where TapMem also improves single-procedure selection, showing that its routing refinement remains effective beyond multi-step procedure orchestration.

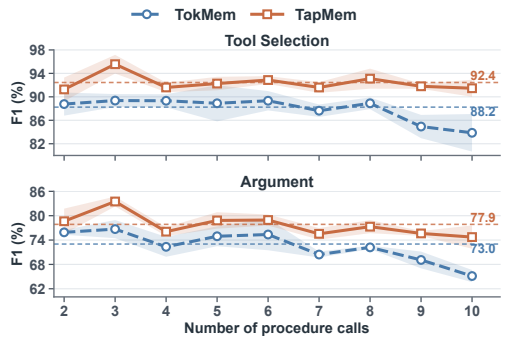


Figure 4: Fine-grained performance comparison

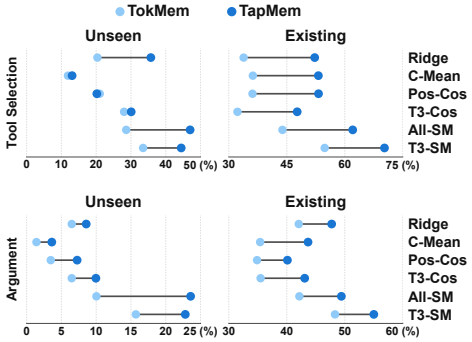


Figure 5: Comparison of cold start methods

Results on the Long-Horizon Setting. (1) *TaskBench.* Figure 3a reports the results on the compositional subset of TaskBench (Shen et al., 2024). Averaged across the 1B, 3B, and 8B backbones, TapMem improves Tool Selection F1 and Argument Generation F1 by 4.7% and 4.1%, respectively, demonstrating its effectiveness on long procedure chains with dependencies explicitly defined by a tool graph. (2) *TRAJECT-Bench.* Figure 3b reports the results on the compositional subset of TRAJECT-Bench (He et al., 2026). TapMem improves Tool Selection F1 and Argument Generation F1 by 5.0% and 5.7% on average across the three backbones, indicating that the gains extend to realistic trajectories with explicit inter-procedure dependencies. (3) *More Procedure Calls.* Figure 4 reports compositional recall results on APIGen with up to 10 procedure calls. TapMem's advantage grows from 2.5% and 2.7% at two calls to 7.6% and 9.6% at ten calls for Tool Selection F1 and Argument Generation F1, respectively. Meanwhile, TokMem degrades more noticeably as the procedure sequence becomes longer, making TapMem's benefit increasingly apparent on extended sequences. Taken together, these results show that TapMem better maintains the quality of procedure selection and argument generation as the procedure sequence becomes longer. Detailed setting and results on long-horizon setting are provided in Appendix D.3.1 and Appendix E.2.1.

Results on the Multi-Turn Execution Setting. We conduct experiments on *InterCode-Bash* (Yang et al., 2023), a multi-turn execution environment in which the model performs Bash operations across multiple interaction steps and receives execution feedback from the environment. The success rate measures complete task execution, while the reward captures partial progress toward the target environment state. As shown in Figure 6, TapMem improves the success rate from 32.50% to 33.33%. This suggests that the benefits of transition-aware procedural memory can extend to multi-turn execution settings. Detailed setting is reported in Appendix D.4.

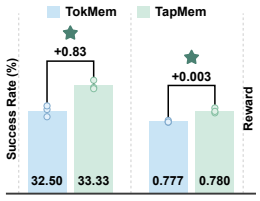


Figure 6: Results on InterCode-Bash.

3.3 Extension to Continual Setting

We further consider a more challenging continual setting, where procedural memory continuously grows as unseen procedures are introduced. Retraining the model whenever an unseen procedure is added is impractical, which leads to a cold-start problem for unseen procedures. As an initial attempt to address this problem, we study six training-free methods from three categories: **semantic-similarity weighting** (T3-SM, T3-Cos, All-SM, and Pos-Cos), **clustering** (C-Mean), and **regression** (Ridge). Detailed methods and results are provided in Appendix D.5 and Appendix E.3.

Figure 5 reports the results on both unseen and existing procedures. We draw the following conclusions: (1) *Cold-start adaptation is a challenging setting for tokenized procedural memory.* Across the six methods, TokMem achieves a maximum Argument Generation F1 of only 15.7% on unseen procedures. (2) *TapMem remains effective in the continual setting.* Across different cold-start methods, TapMem improves procedural recall in most comparisons. Under T3-SM, short for Top-3 Softmax, it improves Tool Selection F1 and Argument Generation F1 by 9.7% and 5.7%, respectively. Detailed results are provided in Appendix E.3.

Table 2: Ablation study on the effectiveness of ETS and TCRA in terms of F1

ETS	TCRA	Tool Selection				Argument			
		2 calls	3 calls	4 calls	Avg.	2 calls	3 calls	4 calls	Avg.
		83.2	82.2	76.4	82.4	67.6	67.6	55.2	67.0
✓		83.6	85.3	75.8	84.0	68.0	70.7	54.8	68.6
✓	✓	88.8	89.6	81.0	**88.8**	74.9	74.1	65.0	**74.0**

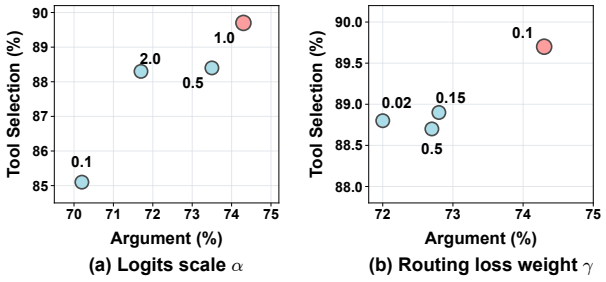


Figure 7: Ablation results on logits scale α and transition-conditioned routing loss weight γ

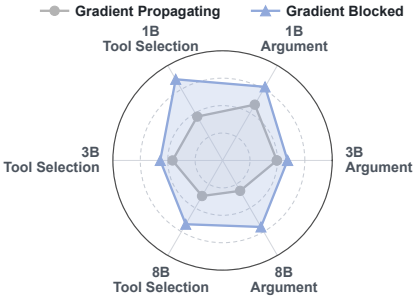


Figure 8: Ablation results on gradient flow. Details are reported in Table 9

3.4 Ablation Studies

We perform several ablation studies to test the effectiveness of our method. Detailed results are provided in Appendix E.4.

Ablation Study on Different Modules. Table 2 present a module ablation of ETS and TCRA to evaluate their contributions to compositional recall. Adding ETS improves Tool Selection F1 by 1.6% and Arguments Generation F1 by 1.6% on the 1B backbones, showing that explicit termination helps identify procedure boundaries. TCRA further improves Tool Selection F1 by up to 4.8% and Arguments Generation F1 by up to 5.4%, demonstrating its advantage on refining transition states into accurate next-procedure routing. More results can be found in Table 8.

Ablation Study on Hyper-Parameters. Figure 7 shows the ablation results for the logits scale α and the transition-conditioned routing loss weight γ on Llama 3.2 1B. For the logit scale, the sequence $0.1 \rightarrow 0.5 \rightarrow 1.0$ leads to steady gains in both Tool Selection F1 and Argument Generation F1, while the scale of 2.0 breaks this trend. This result suggests that a small scale gives weak correction, whereas a large scale reduces the accuracy of argument generation. For the transition-conditioned routing loss weight, both tool selection and argument generation reach their best results at 0.1. This finding suggests that TCRA needs a routing strength that is strong enough to improve memory-token selection but not so strong that it harms normal next-token prediction.

Ablation study on gradient design. Figure 8 shows the results under different gradient designs. For Argument Generation F1, blocked gradient improves performance over gradient propagation, with a gain of nearly 8% on the 8B model. Meanwhile, Tool Selection F1 shows no drop in performance. The result suggests that propagating adapter gradients into token embeddings may distract the model from standard autoregressive learning and weaken its ability to complete the task.

4 Conclusion

This paper studies multi-step procedural orchestration in LLMs and identifies an implicit transition bottleneck in tokenized procedural memory. We propose *TapMem*, a lightweight transition-aware framework that uses ETS to expose procedure boundaries and TCRA to improve next-procedure selection. ETS is implemented with `<EOC>` token, while TCRA corrects only candidate memory-token logits at procedure-selection positions. This design preserves ordinary full-vocabulary generation. Experiments on standard, challenging, multi-step execution and continual settings show improvements over TokMem, highlighting explicit transition modeling as an important point for reliable tokenized procedural memory.

AI USE STATEMENT

We used generative AI tools to aid and polish writing. We have reviewed all AI-assisted work. AI-assisted writing was manually checked and revised by the authors to ensure that the final text accurately reflects the intended technical content and claims. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Sadhana Kumaravel, Matthew Stallone, Rameswar Panda, Yara Rizk, G. P. Shrivatsa Bhargav, Maxwell Crouse, Chulaka Gunasekara, Shajith Ikbal, Sachindra Joshi, Hima Karanam, Vineet Kumar, Asim Munawar, Sumit Neelam, Dinesh Raghu, Udit Sharma, Adriana Meza Soria, Dheeraj Sreedhar, Praveen Venkateswaran, Merve Unuvar, David Cox, Salim Roukos, Luis A. Lastras, and Pavan Kapanipathi. Granite-function calling model: Introducing function calling abilities via multi-task learning of granular tasks. In *Empirical Methods in Natural Language Processing*, pp. 1131–1139, 2024.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, 2023.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens. In *International Conference on Machine Learning*, pp. 2206–2240, 2022.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems*, 2020.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *International Conference on Learning Representations*, 2024.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory. In *European Conference on Artificial Intelligence*, pp. 2993–3000, 2025.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems*, 2023.

Pengfei Du. Memory for autonomous llm agents: Mechanisms, evaluation, and emerging frontiers. *arXiv preprint arXiv:2603.07670*, 2026.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. In *Association for Computational Linguistics*, pp. 17490–17502, 2026.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Empirical Methods in Natural Language Processing*, pp. 10932–10952, 2023.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models. In *Association for Computational Linguistics*, pp. 11143–11156, 2024.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. Retrieval augmented language model pre-training. In *International Conference on Machine Learning*, pp. 3929–3938, 2020.

Dongge Han, Camille Couturier, Daniel Madrigal Díaz, Xuchao Zhang, Victor Rühle, and Saravan Rajmohan. Legomem: Modular procedural memory for multi-agent LLM systems for workflow automation. In *Autonomous Agents and Multiagent Systems*, pp. 3116–3118, 2026.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems*, 2023.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. In *Association for Computational Linguistics*, pp. 6864–6890, 2024.

Pengfei He, Zhenwei Dai, Bing He, Hui Liu, Xianfeng Tang, Hanqing Lu, Juanhui Li, Jiayuan Ding, Subhabrata Mukherjee, Suhang Wang, et al. Traject-bench: a trajectory-aware benchmark for evaluating agentic tool use. In *International Conference on Learning Representations*, pp. 61766–61801, 2026.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*, pp. 2790–2799, 2019.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pp. 9118–9147, 2022.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Empirical Methods in Natural Language Processing*, pp. 6769–6781, 2020.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Empirical Methods in Natural Language Processing*, pp. 3045–3059, 2021.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems*, 2020.

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Association for Computational Linguistics*, pp. 4582–4597, 2021.

Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pp. 74–81. Association for Computational Linguistics, 2004.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *International Conference on Language Resources and Evaluation*, 2018.

Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks. In *Association for Computational Linguistics*, pp. 61–68, 2022.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, pp. 52989–53046, 2024a.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh Murthy, Liangwei Yang, Silvio Savarese, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, and Caiming Xiong. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. In *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems*, 2024b.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. In *Nations of the Americas Chapter of the Association for Computational Linguistics*, pp. 1160–1183, 2025.

Ali Modarressi, Ayyoob Imani, Mohsen Fayyaz, and Hinrich Schütze. Ret-llm: Towards a general read-write memory for large language models. *arXiv preprint arXiv:2305.14322*, 2023.

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.

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

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): from tool use to agentic evaluation of large language models. In *International Conference on Machine Learning*, 2025.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, 2024.

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Empirical Methods in Natural Language Processing*, pp. 3982–3992, 2019.

Hayley Ross, Ameya Sunil Mahabaleshwarkar, and Yoshi Suhara. When2call: When (not) to call tools. In *Nations of the Americas Chapter of the Association for Computational Linguistics*, pp. 3391–3409, 2025.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems*, 2023.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving AI tasks with chatgpt and its friends in huggingface. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems*, 2023.

Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. Taskbench: Benchmarking large language models for task automation. In *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems*, 2024.

Qwen Team. Qwen3.5: Towards native multimodal agents, 2026. URL https://qwen.ai/blog?id=qwen3.5.

Renxi Wang, Xudong Han, Lei Ji, Shu Wang, Timothy Baldwin, and Haonan Li. Toolgen: Unified tool retrieval and calling via generation. In *International Conference on Learning Representations*, 2025a.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems*, 2020.

Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Atharva Naik, Arjun Ashok, Arut Selvan Dhanasekaran, Anjana Arunkumar, David Stap, Eshaan Pathak, Giannis Karamanolakis, Haizhi Gary Lai, Ishan Purohit, Ishani Mondal, Jacob Anderson, Kirby Kuznia, Krima Doshi, Kuntal Kumar Pal, Maitreya Patel, Mehrad Moradshahi, Mihir Parmar, Mirali Purohit, Neeraj Varshney, Phani Rohitha Kaza, Pulkit Verma, Ravsehaj Singh Puri, Rushang Karia, Savan Doshi, Shailaja Keyur Sampat, Siddhartha Mishra, Sujan Reddy A, Sumanta Patro, Tanay Dixit, and Xudong Shen. Super-naturalinstructions: Generalization via declarative instructions on 1600+ NLP tasks. In *Empirical Methods in Natural Language Processing*, pp. 5085–5109, 2022.

Yu Wang, Yifan Gao, Xiusi Chen, Haoming Jiang, Shiyang Li, Jingfeng Yang, Qingyu Yin, Zheng Li, Xian Li, Bing Yin, Jingbo Shang, and Julian McAuley. MEMORYLLM: towards self-updatable large language models. In *International Conference on Machine Learning*, pp. 50453–50466, 2024.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems*, 2022.

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

Zijun Wu, Yongchang Hao, and Lili Mou. TokMem: One-token procedural memory for large language models. In *International Conference on Learning Representations*, pp. 32460–32480, 2026.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems*, 2025.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems*, 2023.

Qwen: An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

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

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Kan Ren, Dongsheng Li, and Deqing Yang. EASYTOOL: enhancing llm-based agents with concise tool instruction. In *Nations of the Americas Chapter of the Association for Computational Linguistics*, pp. 951–972, 2025.

Guibin Zhang, Muxin Fu, and Shuicheng Yan. Memgen: Weaving generative latent memory for self-evolving agents. In *International Conference on Learning Representations*, volume 2026, pp. 22555–22588, 2026.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, 2024.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, et al. Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *arXiv preprint arXiv:2305.17144*, 2023.

CONTENTS

A. Related Work **15**

A.1. Memory-Augmented Methods for LLMs 15

A.2. Tokenized Procedural Memory 15

B. Additional Examples for TapMem **15**

C. Algorithm of TapMem **15**

D. Additional Experimental Setting **15**

D.1. Details of Competitors 18

D.2. Detailed Standard Settings 19

D.3. Detailed Challenging Setting 20

D.4. Detailed Multi-Turn Execution Setting 22

D.5. Detailed Continual Setting 23

E. Additional Experimental Results **25**

E.1. Expanded Version of Results on Standard Setting 25

E.2. Expanded Version of Results of Long-Horizon Setting 27

E.3. Expanded Version of Results on Continual Setting 27

E.4. Expanded Version of Results on Ablation Study 28

E.5. Analysis of Correct Prefix Length 28

F. Comparison of Parameter Count **28**

A Related Work

A.1 Memory-Augmented Methods for LLMs

Memory augmentation allows LLMs to use task-relevant knowledge beyond the immediate input (Wu et al., 2025; Du, 2026). In in-context learning, demonstrations, instructions, and reasoning traces can serve as temporary textual memory in the prompt (Brown et al., 2020; Wei et al., 2022; Yao et al., 2023; Yuan et al., 2025). In retrieval-augmented generation, an external memory store provides relevant documents, examples, or tool descriptions at inference time (Karpukhin et al., 2020; Lewis et al., 2020; Borgeaud et al., 2022; Patil et al., 2024; Qin et al., 2024; Guu et al., 2020; Modarressi et al., 2023). Recent work on agent-based memory systems further extends text-based memory methods(Xu et al., 2025; Chhikara et al., 2025; Packer et al., 2023; Zhu et al., 2023; Wang et al., 2025b; Zhang et al., 2026; Shen et al., 2023). Some studies also place reusable behavior in trainable parameters or continuous representations, including adapters, soft prompts, prefix tuning, P-Tuning, and LoRA (Houlsby et al., 2019; Li & Liang, 2021; Lester et al., 2021; Liu et al., 2022; Hu et al., 2022; Wang et al., 2024). Although these designs provide useful forms of memory, they still have limitations for procedural reuse. Textual and retrieval-based memory requires extra context construction at each use (Wu et al., 2025), while behavior stored in parameters is hard to inspect and edit, and it lacks a clear discrete call interface. More importantly, it remains unclear how these memory mechanisms should guide a LLM from one reusable procedure to another within a single autoregressive generation, leaving multi-step procedural orchestration under-specified (Fang et al., 2026).

A.2 Tokenized Procedural Memory

Tokenized procedural memory (Wu et al., 2026; Hao et al., 2023; Wang et al., 2025a) provides a more compact formulation, where each reusable procedure is represented as a discrete trainable memory token. A representative example is TokMem (Wu et al., 2026), which compiles a procedure into a single memory token. The corresponding behavior is then called through standard language-model generation. Long procedure descriptions no longer need to be repeatedly added in the context, and procedure recall remains compatible with autoregressive decoding. However, TokMem still has a key limitation in compositional settings. When multiple procedures needed in sequence (Huang et al., 2022; Schick et al., 2023), the model has to decide when the current memory-guided span should end and which procedure should be selected next. In TokMem, the transition between procedures is left to standard next-token prediction, so procedure termination and next-procedure selection remain implicit and coupled. This makes procedural chaining less stable, especially when the number of procedure calls increases. TapMem is motivated by this missing transition mechanism: it keeps the memory-token form, while using explicit termination signals and transition-aware routing adapter to make procedure transitions explicit.

B Additional Examples for TapMem

In Figure 9, we present a real tool-use example. TokMem selects an incorrect procedure `<binary_search>` and terminates generation prematurely, whereas TapMem selects the correct procedure and generates all arguments accurately. The example shows that ETS, instantiated by `<EOC>`, and routing refinement can explicitly strengthen transition modeling, thereby improving the accuracy of procedure selection and argument generation.

C Algorithm of TapMem

For clarity, we provide the pseudocode of our *Transition-Aware Procedural Memory* in Algorithm 1 and Algorithm 2.

D Additional Experimental Setting

We organize the experiments in this paper into four settings, which together cover both the recall and the execution of procedural memory. (1) *Standard setting* follows TokMem (Wu et al., 2026)

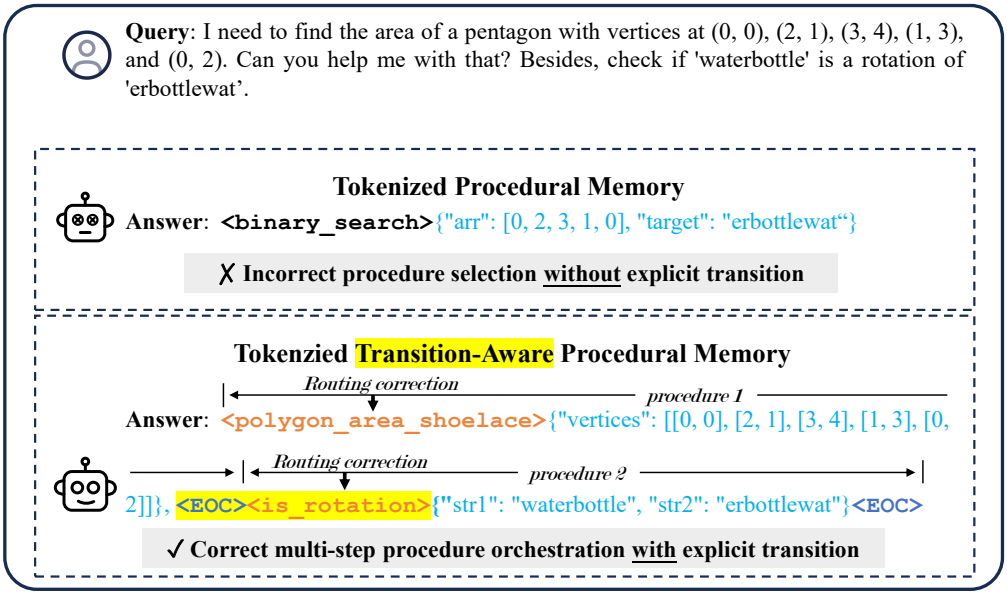


Figure 9: A real example of compositional procedure recall

Algorithm 1 Training TapMem

Input: Frozen LLM f_θ; training data $\mathcal{D}$; memory-token set $\mathcal{V}_M = \{m_k\}_{k=1}^{K}$; <EOC>; adapter g_ϕ; α, γ.
Output: $\mathbf{E}$, $\mathbf{e}_{\text{eoc}}$, and g_ϕ.

Initialize $\mathbf{E}$, $\mathbf{e}_{\text{eoc}}$, and g_ϕ.
for each $(\mathbf{q}, \{m^{(j)}, \mathbf{a}^{(j)}\}_{j=1}^{J}) \in \mathcal{D}$ **do**
  $\mathbf{y}' \leftarrow [\mathbf{q}, m^{(1)}, \mathbf{a}^{(1)}, \texttt{<EOC>}, \ldots, m^{(J)}, \mathbf{a}^{(J)}, \texttt{<EOC>}]$
  $\mathcal{B} \leftarrow \{b \mid y'_b = \texttt{<EOC>},\ y'_{b+1} \in \mathcal{V}_M\}, \quad \mathcal{R} \leftarrow \{|\mathbf{q}|+1\} \cup \{b+1 \mid b \in \mathcal{B}\}$
  Run f_θ on $\mathbf{y}'$ to obtain $\{\mathbf{h}_{t-1}, \mathbf{z}_t^{\text{LM}}\}$, and set $\widetilde{\mathbf{z}}_t \leftarrow \mathbf{z}_t^{\text{LM}}$.
  ▷ *Transition-conditioned routing*
  for each $r \in \mathcal{R}$ **do**
    $\boldsymbol{\pi}_r \leftarrow \text{softmax}(g_\phi(\text{stopgrad}(\mathbf{h}_{r-1})))$
    $\widetilde{\mathbf{z}}_{r,\mathcal{V}_M} \leftarrow \mathbf{z}^{\text{LM}}_{r,\mathcal{V}_M} + \alpha\,(\log \boldsymbol{\pi}_r - \log(1/K))$
  end for
  ▷ *Joint optimization*
  $\mathcal{L}_{\text{AR}} \leftarrow -\sum_{t=|\mathbf{q}|+1}^{|\mathbf{y}'|} \log\,[\text{softmax}(\widetilde{\mathbf{z}}_t)]_{y'_t}$
  $\mathcal{L}_{\text{route}} \leftarrow -\frac{1}{|\mathcal{R}|}\sum_{r\in\mathcal{R}} \log \pi_r(k_r), \quad y'_r = m_{k_r}$
  $\mathcal{L} \leftarrow \mathcal{L}_{\text{AR}} + \gamma\mathcal{L}_{\text{route}}$
  Update $\mathbf{E}$, $\mathbf{e}_{\text{eoc}}$, and ϕ using $\mathcal{L}$.
end for
return $\mathbf{E}$, $\mathbf{e}_{\text{eoc}}$, and g_ϕ.

and contains two recall protocols: compositional recall on APIGen, where a single request needs 2–4 procedure calls and is evaluated on Llama 3.2 1B/3B, Llama 3.1 8B, Qwen 3.5 4B/9B, and atomic recall on SNI, where each input needs exactly one procedure and is evaluated on Qwen 2.5 0.5B, Llama 3.2 3B, and Llama 3.1 8B. (2) *Long-horizon setting* keeps the compositional recall protocol but increases the number of procedure calls, and contains three experiments: compositional recall on TaskBench, compositional recall on TRAJECT-Bench, and an APIGen extension with up

Algorithm 2 Inference with TapMem

Input: Frozen LLM f_θ; trained $\mathbf{E}$, $\mathbf{e}_{\text{eoc}}$, and g_ϕ; query $\mathbf{q}$; memory-token set $\mathcal{V}_M = \{m_k\}_{k=1}^{K}$; logit scale α; maximum sequence length T.
Output: Generated sequence $\hat{\mathbf{y}}$.

$\hat{\mathbf{y}} \leftarrow \mathbf{q}$
for $r = |\mathbf{q}| + 1$ to T **do**
  Run f_θ on $\hat{\mathbf{y}}_{<r}$ to obtain $\mathbf{h}_{r-1}$ and $\mathbf{z}_r^{\text{LM}}$.
  $\widetilde{\mathbf{z}}_r \leftarrow \mathbf{z}_r^{\text{LM}}$
  ▷ ***Boundary-conditioned routing***
  if $r = |\mathbf{q}| + 1$ **or** $\hat{y}_{r-1} =$ <EOC> **then**
    $\mathbf{z}_r^{\text{TCRA}} \leftarrow g_\phi(\mathbf{h}_{r-1})$
    $\boldsymbol{\pi}_r \leftarrow \text{softmax}(\mathbf{z}_r^{\text{TCRA}})$
    for $k = 1, \ldots, K$ **do**
      $\Delta_r(k) \leftarrow \log \pi_r(k) - \log(1/K)$
      $\widetilde{z}_{r,m_k} \leftarrow z_{r,m_k}^{\text{LM}} + \alpha \Delta_r(k)$
    end for
  end if
  ▷ ***Full-vocabulary autoregressive decoding***
  Decode $\hat{y}_r$ from $\text{softmax}(\widetilde{\mathbf{z}}_r)$ over $\widetilde{\mathcal{V}}$.
  Append $\hat{y}_r$ to $\hat{\mathbf{y}}$.
  if $\hat{y}_r =$ <EOS> **then**
    return $\hat{\mathbf{y}}$.
  end if
end for
return $\hat{\mathbf{y}}$.

to 10 procedure calls. (3) *Multi-turn execution setting* moves from generation within one response to interaction with an environment, and contains one experiment on InterCode-Bash, where the model issues Bash commands, receives execution feedback, and is measured by success rate and reward. (4) *Continual setting* studies memory expansion, and contains one experiment that adds 10 unseen procedures to the 50 existing ones under six training-free cold-start methods. Except for the multi-turn execution setting, which reports success rate and reward, all other settings report Tool Selection F1 and Argument Generation F1.

Implementation Details. In the main experiments, the training is conducted for 3 epochs on synthetic data from APIGen (Liu et al., 2024b) for compositional recall. Our main experiments use Llama 3.2 1B/3B and Llama 3.1 8B (Grattafiori et al., 2024), with additional evaluations on Qwen 3.5 4B/9B (Team, 2026) to examine cross-family generalization. The atomic-recall experiments use Qwen 2.5 0.5B (Yang et al., 2025), Llama 3.2 3B, and Llama 3.1 8B. Greedy decoding is employed for all evaluations. The main results are reported as the average over three random seeds. The learning rate for memory embedding is 5×10^{-3}, 4×10^{-3}, and 5×10^{-3} for the 1B, 3B, and 8B backbones. We adopt *Adam with Weight Decay (AdamW)* (Kingma & Ba, 2015; Loshchilov & Hutter, 2019) optimizer with a weight decay of 0.01. For all experiments, TokMem (Wu et al., 2026) serves as the baseline of tokenized procedural memory. The TCRA adapter is implemented as a linear projection layer. All experiments are conducted on a single NVIDIA-A100 80GB GPU using mixed-precision bfloat16 training.

The following itemized descriptions detail the evaluation protocol and the capability assessed in each setting:

- **Standard setting.** The standard setting evaluates the basic reliability of procedural memory under the recall protocols considered by TokMem (Wu et al., 2026). We distinguish between compositional recall and atomic recall as they isolate different aspects of procedural memory.
  - *Compositional recall.* Compositional recall on APIGen is designed to assess whether procedural memory can support multi-step procedure generation when a single user request requires several procedures to be coordinated within one generation. Unlike atomic recall, this setting requires the model to correctly coordinate multiple procedures. It therefore provides evidence for the

model's ability to perform reliable multi-procedure orchestration, and for the usefulness of TapMem's explicit procedure-boundary representation in supporting such compositional use of procedural memory.

  - *Atomic recall.* Atomic recall evaluates the single-procedure case, in which the model selects one procedural memory before generating the corresponding response. By removing the need to coordinate multiple procedures, this setting isolates the reliability of basic memory-token routing and procedure-conditioned generation. Improvements in this setting indicate that the proposed routing refinement is not only useful when modeling transitions in compositional trajectories, but also improves the selection of an individual procedural memory in isolation. Thus, this setting assesses the reliability of procedural recall independently of multi-step orchestration.

- **Long-horizon setting.** Beyond the short compositional sequences considered in the standard setting, the long-horizon setting examines longer procedure sequences with stronger dependencies. As the length of procedure chaining increases, an early routing error can propagate to later calls, requiring the model to preserve relevant procedural information and make reliable procedure selections throughout the trajectory. Accordingly, the long-horizon setting asks whether the method can preserve the quality of procedure selection and generation as the sequence grows and intermediate errors become more consequential. In this way, the setting assesses whether the method's effectiveness extends from short compositional cases to longer and more demanding procedure sequences.
- **Multi-turn execution setting.** Multi-turn execution on InterCode-Bash evaluates procedural memory in an interactive environment where multiple procedure calls are interleaved with environmental feedback. Successful execution requires the model to preserve the task objective while adapting procedure selection to changes in the execution context. The evaluation therefore focuses on state-dependent procedure orchestration and feedback-conditioned adaptation, rather than procedural recall within a fixed generation.
- **Continual setting.** The continual setting evaluates whether procedural memory can incorporate procedures unseen during training while retaining reliable recall of existing procedures. Under a cold-start protocol, the memory-token embedding and corresponding TCRA parameters for each unseen procedure are initialized from the existing procedural memory without demonstrations or gradient updates, and performance is evaluated separately on unseen and existing procedures. This setting therefore measures training-free memory expansion and the robustness of procedure selection and generation as the set of stored procedures grows.

Taken together, these settings evaluate procedural memory at increasing levels of difficulty: reliable routing of an individual procedure, coordination of multiple procedures within one generation, reliable procedure selection and generation over longer sequences, feedback-conditioned procedure orchestration across multiple interaction steps, and cold-start extension to unseen procedures while preserving recall of existing procedures.

D.1 Details of Competitors

Here, we provide a detailed summary of the representative baselines spanning textualized context engineering, retrieval-augmented memory, parameter-efficient fine-tuning, and tokenized procedural memory methods.

- **Textualized context engineering.** Across all settings for compositional recall, the *ICL* baseline places the available tool descriptions in the prompt and lets the model produce the sequence of function-call arguments directly (Brown et al., 2020; Patil et al., 2024; Qin et al., 2024). As for atomic recall, the *Base* baseline directly provides the task instruction in text and asks the model to generate the answer without any trainable procedural memory (Wu et al., 2026).
- **Retrieval-augmented textual memory.** Across all settings for compositional recall, *RAG* indexes the tool descriptions and retrieves a small set of candidate tools, e.g., top-k tools provided by benchmark itself, which are then placed in the prompt for generation (Lewis et al., 2020; Karpukhin et al., 2020; Patil et al., 2024; Qin et al., 2024). As for atomic recall, *RAG* builds a non-parametric retrieval index from Natural Instructions task demonstrations and retrieves the most similar demonstration according to the input query (Lewis et al., 2020; Karpukhin et al., 2020; Wang et al., 2022).

- **Parameter-efficient fine-tuning.** We include *LoRA* as a parameter-efficient learning baseline for compositional tool invocation. LoRA adapts the frozen backbone through a small set of low-rank update matrices inserted into selected attention projections (Hu et al., 2022). This baseline evaluates whether reusable procedures can be captured by lightweight parametric adaptation alone. The LoRA learning rate is 5×10^{-5} for Llama 3.2 1B/3B and 8×10^{-6} for Llama 3.1 8B.

- **Tokenized Procedural Memory.** We use *TokMem* as the baseline of tokenized procedural memory. It represents each procedure as a trainable token and invokes procedures through autoregressive token generation (Wu et al., 2026).

D.2 Detailed Standard Settings

Following TokMem (Wu et al., 2026), we conduct a variety of experiments under the standard setting, including compositional recall on APIGen with both the Llama and Qwen model families, as well as atomic recall on SNI.

D.2.1 Compositional Recall on APIGen

Dataset For compositional recall, we use a multi-tool calling benchmark synthesized from APIGen (Liu et al., 2024b). APIGen is an automated data generation pipeline for constructing verifiable and diverse function-calling datasets. It collects 3,673 executable APIs across 21 categories and uses them to synthesize natural-language query–response pairs in a scalable and structured way. The released APIGen dataset contains 60,000 high-quality entries and covers multiple query styles, including simple, multiple, parallel, and parallel-multiple function-calling cases, making it a suitable resource for evaluating and training LLMs on tool selection and argument generation.

Data Construction Following TokMem (Wu et al., 2026), we remove web-search-related tools from the dataset and select the 100 tools with the most samples. We use the first 50 tools to construct the training dataset for the adaptation stage and the remaining 50 tools to construct the dataset for the main training stage of TapMem. For TapMem with adaptation, the training data used in the second stage is the same as the training data used by TapMem without adaptation. Each tool contains 50 query-response pairs, which are treated as atomic samples and included as part of the training data. We further use these data to synthesize compositional data by randomly sampling atomic samples and combining them into compositional samples, where conjunctions are inserted into the queries. Each compositional sample contains at least 2 calls and at most 4 calls. All samples in the test set are compositional samples. In total, 2,500 atomic samples and 2,500 compositional samples are included in the training set, while 500 samples are included in the test set.

Metrics For compositional function-calling, we report *Tool Selection F1* and *Arguments Generation F1*. Tool Selection F1 evaluates the precision and recall of predicted procedures with respect to the reference calls, while Arguments Generation F1 requires both the procedure and its generated arguments to be correct. Both metrics are reported as the harmonic mean of precision and recall.

Transition Error We evaluate transition errors at both the token level and the sample level. At the token level, each gold transition step is considered separately. The first procedure in an example is treated as the initial decision, whereas each subsequent procedure is treated as a transition decision because it requires the model to move from one procedure to the next. A token-level transition error occurs when the model predicts an incorrect procedure at any subsequent procedure-selection step. If the model stops early and misses a later gold procedure, that step is also counted as incorrect. Extra predicted procedures beyond the gold sequence are ignored under this metric. At the sample level, we examine whether an example contains any transition error. If any subsequent procedure-selection step is predicted incorrectly, the whole example is counted as having a transition error. Therefore, the sample-level metric measures how often a multi-procedure example contains at least one transition failure, whereas the token-level metric measures how many individual transition decisions are incorrect.

Table 3: **Atomic recall results on the SNI dataset.** This setting contains a single procedural memory selection per example. Results for TokMem and TapMem are reported as the mean followed by the sample standard deviation across three seeds. For the deterministic Base and RAG baselines, the standard deviation is not applicable. "–" denotes that the metric is not applicable. The champion and the runner-up within each model are highlighted in **bold** and underline, respectively.

Model	Metric	Base	RAG	TokMem	TapMem
Qwen 2.5 0.5B	Routing Accuracy	–	–	90.60 (±0.13)	**93.85** (±0.13)
	Rouge-L	40.45	33.43	46.68 (±0.06)	**46.75** (±0.13)
Llama 3.2 3B	Routing Accuracy	–	–	95.58 (±0.19)	**97.15** (±0.14)
	Rouge-L	20.15	40.25	51.96 (±0.50)	**52.20** (±0.45)
Llama 3.1 8B	Routing Accuracy	–	–	97.08 (±0.02)	**97.68** (±0.11)
	Rouge-L	21.82	42.33	57.66 (±0.16)	**57.89** (±0.21)

D.2.2 Compositional Recall on More Backbones

We additionally evaluate TapMem on the APIGen compositional recall benchmark with more backbones, namely Qwen 3.5 4B and Qwen 3.5 9B (Team, 2026), in order to examine whether the gains observed on the Llama family transfer across model families. This setting follows the APIGen compositional recall protocol described in Section D.2.1 and adopts the same configuration.

D.2.3 Atomic Recall on SNI

Dataset Super-Natural Instructions (SNI) (Wang et al., 2022) is a large-scale collection of instruction-formatted NLP tasks. Each task consists of a natural-language definition, positive and negative examples, and multiple input-output instances. The dataset is commonly used to evaluate instruction following and cross-task generalization. It covers 1,616 tasks across 76 task types, where each task is treated as a reusable procedure and represented by a procedural memory token. Each input is associated with exactly one target procedure, so the model only needs to select a single memory token before generating the corresponding answer.

Metrics For atomic memory recall, we use procedural memory token *Routing Accuracy* to measure the correctness of memory-token selection and *ROUGE-L* (Lin, 2004) to evaluate the quality of the generated textual response.

Data Construction Following the process as TokMem, we first remove cross-lingual tasks from the SNI dataset. From the remaining tasks, we sample 700 tasks, with 80 training samples, 10 validation samples, and 20 test samples drawn from each task.

D.3 Detailed Challenging Setting

D.3.1 Compositional recall on TaskBench and TRAJECT-Bench

Beyond the standard setting following TokMem, we further conduct two additional experiments on TaskBench and TRAJECT-Bench and evaluate our method with more tool calls on APIGen. Following the APIGen setting, we use the same evaluation metrics for both benchmarks: Tool Selection F1 and Argument Generation F1.

Benchmarks TaskBench (Shen et al., 2024) is a benchmark for evaluating LLMs on multi-step planning. It organizes tools as a tool graph, where nodes represent procedures and directed edges encode dependencies between them. Tasks are constructed from different graph structures, making it suitable for evaluating procedure orchestration under explicit dependency constraints. TRAJECT-Bench (He et al., 2026) evaluates LLMs on realistic trajectories across diverse application domains. It includes multi-step trajectories with varying lengths and dependency structures. We use its sequential trajectories to evaluate long-horizon procedure orchestration with explicit inter-procedure dependencies.

Table 4: **Fine-grained** atomic recall results for the top 20 categories by task count. Each category is shown as a paired comparison between TokMem and our method; Δ is computed by subtracting the displayed TokMem value from the displayed value of our method.

Category	Tasks	Method	Routing Acc (%)	Δ Acc	Rouge-L	Δ Rouge-L
Question Answering	86	TokMem	75.64	+5.19	33.21	+0.30
		Ours	**80.83**		**33.51**	
Program Execution	71	TokMem	92.44	+1.93	31.27	+2.04
		Ours	**94.37**		**33.31**	
Question Generation	44	TokMem	**85.80**	-2.96	24.49	+0.31
		Ours	82.84		**24.80**	
Sentiment Analysis	43	TokMem	97.09	+1.28	**69.00**	-2.75
		Ours	**98.37**		66.25	
Text Matching	32	TokMem	93.13	+0.41	**74.90**	-0.49
		Ours	**93.54**		74.41	
Text Categorization	30	TokMem	95.89	+1.89	49.91	+1.11
		Ours	**97.78**		**51.02**	
Toxic Language Detection	23	TokMem	97.54	+1.23	**58.23**	-1.48
		Ours	**98.77**		56.75	
Commonsense Classification	22	TokMem	99.55	+0.37	**53.01**	-0.28
		Ours	**99.92**		52.73	
Information Extraction	18	TokMem	83.43	+1.01	**43.96**	-0.95
		Ours	**84.44**		43.01	
Textual Entailment	18	TokMem	**90.46**	-0.74	**50.12**	-0.95
		Ours	89.72		49.17	
Wrong Candidate Generation	16	TokMem	78.33	+4.38	21.91	+2.59
		Ours	**82.71**		**24.50**	
Sentence Composition	14	TokMem	80.71	+5.48	36.43	+0.50
		Ours	**86.19**		**36.93**	
Text Completion	14	TokMem	91.43	+0.36	25.92	+2.09
		Ours	**91.79**		**28.01**	
Title Generation	14	TokMem	76.55	+3.09	26.52	+2.50
		Ours	**79.64**		**29.02**	
Misc.	12	TokMem	63.89	+4.17	15.34	+0.39
		Ours	**68.06**		**15.73**	
Question Understanding	12	TokMem	**94.72**	-0.28	43.32	+0.99
		Ours	94.44		**44.31**	
Answerability Classification	10	TokMem	98.50	+1.00	70.39	+1.64
		Ours	**99.50**		**72.03**	
Dialogue Generation	10	TokMem	88.50	+0.50	21.94	+0.34
		Ours	**89.00**		**22.28**	
Language Identification	9	TokMem	92.04	+4.44	60.37	+6.93
		Ours	**96.48**		**67.30**	
Summarization	9	TokMem	**73.33**	-5.18	23.37	+0.07
		Ours	68.15		**23.44**	

Data Construction For TaskBench, we retain the *node* and *chain* examples from the DailyLife subset. Each node example contains one procedure, while each chain example contains several dependent procedures. For each chain example, we arrange the procedures in `task_nodes` using the dependency edges in `task_links`. The resulting order serves as the ground truth procedure sequence. As TaskBench contains repeated generated requests, so we keep examples with the same request in one data partition. Finally, we obtain 3,234 training examples and 809 test examples over 40 procedures. For TRAJECT-Bench, we select the sequential subset from the original data, remove invalid or incomplete tool steps while preserving the relative order of the remaining steps. We then iteratively filter the data so that each procedure occurs at least 12 times. Each retained trajectory is converted into an example consisting of a user request, an ordered procedure sequence, and the corresponding arguments, with minor formatting modifications to adapt the data to our setting.

D.3.2 Compositional Recall with More Tool Calls

For compositional recall with more tool calls, we further extend the synthetic dataset by using the same samples to construct longer samples. Each sample contains up to 10 calls. The training set

Table 5: Additional compositional recall results using **Qwen 3.5 4B and 9B** on the **APIGen** dataset in terms of F1. † denotes training-free methods.

Method	Tool Selection				Argument			
	2 calls	3 calls	4 calls	Avg.	2 calls	3 calls	4 calls	Avg.
Qwen 3.5 4B								
ICL†	75.9	70.9	70.0	73.2	64.2	61.8	62.9	63.0
RAG†	87.2	83.4	87.2	85.3	67.2	62.5	65.3	64.8
TokMem	87.4 (±1.3)	84.0 (±0.5)	75.9 (±7.9)	85.2 (±0.9)	59.6 (±1.0)	56.2 (±1.9)	44.4 (±3.0)	57.2 (±1.4)
TapMem	93.6 (±3.0)	91.6 (±3.5)	91.9 (±3.4)	**92.6** (±3.3)	69.8 (±8.2)	71.3 (±6.9)	67.5 (±7.5)	**70.4** (±7.5)
TokMem + adaptation	98.0 (±1.6)	97.0 (±0.8)	95.2 (±1.2)	97.4 (±1.1)	77.9 (±2.3)	77.3 (±1.7)	74.7 (±4.0)	77.4 (±2.0)
TapMem + adaptation	98.6 (±0.4)	98.7 (±0.6)	98.1 (±1.1)	**98.6** (±0.5)	83.9 (±2.7)	83.6 (±1.5)	79.4 (±1.8)	**83.5** (±2.0)
Qwen 3.5 9B								
ICL†	77.3	74.2	73.9	75.6	68.2	67.3	72.9	68.0
RAG†	87.5	81.7	79.7	84.3	71.5	67.2	62.2	69.0
TokMem	93.9 (±2.1)	89.8 (±1.4)	89.3 (±4.8)	91.7 (±1.5)	71.9 (±4.6)	67.4 (±3.6)	62.6 (±6.7)	69.3 (±3.8)
TapMem	96.0 (±1.3)	95.1 (±0.7)	93.0 (±4.7)	**95.4** (±0.6)	79.0 (±2.2)	79.3 (±0.9)	71.2 (±9.7)	**78.8** (±1.7)
TokMem + adaptation	98.5 (±0.5)	98.6 (±0.7)	97.8 (±0.4)	98.5 (±0.6)	82.7 (±1.9)	85.3 (±2.6)	75.8 (±0.8)	**83.6** (±2.1)
TapMem + adaptation	99.3 (±0.5)	99.3 (±0.2)	99.8 (±0.3)	**99.4** (±0.2)	82.3 (±2.5)	84.4 (±2.5)	81.3 (±2.0)	83.2 (±2.3)

Table 6: Detailed compositional recall results on **TaskBench and TRAJECT-Bench** in terms of F1. † denotes training-free methods.

Benchmark	Model	Method	Tool Selection (%)	Argument (%)
TaskBench	*Llama 3.2 1B*	ICL†	2.7	3.1
		RAG†	8.1	8.2
		TokMem	88.9(±0.9)	66.6(±1.5)
		TapMem	**93.8**(±0.8)	**70.7**(±1.1)
	Llama 3.2 3B	ICL†	46.2	43.0
		RAG†	60.5	57.0
		TokMem	85.0(±1.0)	64.5(±1.1)
		TapMem	**91.2**(±0.6)	**69.5**(±1.1)
	Llama 3.1 8B	ICL†	62.9	63.4
		RAG†	67.6	65.2
		TokMem	87.3(±0.7)	66.2(±0.8)
		TapMem	**90.3**(±0.4)	**69.4**(±1.0)
TRAJECT-Bench	*Llama 3.2 1B*	ICL†	24.0	1.8
		RAG†	49.5	4.3
		TokMem	76.7(±1.6)	45.9(±5.9)
		TapMem	**81.9**(±2.5)	**52.6**(±6.3)
	Llama 3.2 3B	ICL†	68.2	25.7
		RAG†	69.9	33.6
		TokMem	69.3(±6.8)	40.2(±9.3)
		TapMem	**73.0**(±7.4)	**45.5**(±9.1)
	Llama 3.1 8B	ICL†	86.3	52.7
		RAG†	79.9	45.4
		TokMem	80.8(±2.1)	54.6(±2.7)
		TapMem	**86.8**(±0.3)	**59.7**(±3.7)

includes 2,500 atomic samples and 5,500 compositional samples. Among the compositional samples, the proportions of samples with different numbers of tool calls are kept consistent.

D.4 Detailed Multi-Turn Execution Setting

Dataset *InterCode-Bash* (Yang et al., 2023) is an interactive coding benchmark in which a model solves Bash-based tasks by repeatedly issuing shell commands and receiving execution feedback from an environment. Each task consists of a natural-language instruction, an initial execution environment, and a target state that determines whether the task has been completed successfully. We use all 200 official InterCode-Bash tasks and define 62 fixed procedures covering the available

shell operations. *NL2Bash* (Lin et al., 2018) is a dataset for natural-language-to-Bash translation, consisting of natural-language descriptions paired with corresponding Bash commands. We use NL2Bash to construct the training data for procedural memory.

Data Construction Since InterCode-Bash is derived from NL2Bash, we construct the training data from NL2Bash and remove all source groups that overlap with or closely resemble the 200 official InterCode-Bash tasks. We then parse the remaining commands with `bashlex` to find the positions where a command can be safely split. Each procedure corresponds to one shell utility. We keep the utilities that occur frequently enough. The resulting data yields 62 procedures and 5,259 unique training examples. Each example pairs one instruction with its command, and the command itself is the training target. The procedural memory embedding is trained on this data and applied without modification in the multi-turn execution setting.

Interaction Protocol At each interaction step, the model receives the original task instruction together with the execution feedback produced by the previous step, and then generates the next Bash command. The command is executed in the environment before the resulting feedback is returned to the model. We allow at most two command-execution interactions per task, implemented with `max_turns=2`. Greedy decoding is used throughout evaluation. A task is terminated when the model produces the right response or when the interaction budget is exhausted.

Metrics We report task *success rate* and *reward* as the benchmark does. The success rate is the proportion of evaluation tasks whose final environment state matches the target state. The reward is the score that the environment assigns to the current environment state after every executed command. It is computed by comparing the state against the target state in terms of file-system changes, file contents, and command output; it ranges from 0 to 1 and reaches 1 only when the state is completely correct, so it measures how much progress a command has made toward the target state. Because a task may make partial progress without being solved, we report the average of the maximum reward attained along the interaction trajectory.

D.5 Detailed Continual Setting

Experimental setting. All continual experiments start from the same Llama 3.1 8B checkpoint used in the standard compositional recall setting, where the first 50 procedures form the existing procedural memory. We then add 10 procedures that are never seen during training and evaluate on a mixed test set that contains both existing and unseen procedures, reporting Tool Selection F1 and Argument Generation F1 on the full set and separately on the two subsets. Throughout this process, we collect no demonstrations and perform no gradient updates, and the backbone together with all existing memory-token embeddings and TCRA parameters are kept frozen. Only the memory-token embedding and TCRA weight of an unseen procedure are synthesized. The same synthesis protocol and the same six cold-start methods are applied to TokMem and TapMem, so the comparison isolates how each base method accommodates an unseen procedure.

We define $\mathcal{P}_{\text{exist}} = \{p_i\}_{i=1}^{K}$ for the existing procedures and $\mathcal{P}_{\text{unseen}} = \{p_u\}_{u=1}^{U}$ for the unseen ones, then represent each procedure by its name and description. Using Sentence-BERT (Reimers & Gurevych, 2019; Wang et al., 2020) as the frozen sentence encoder q_ψ, we encode this information and normalize the resulting representation:

$$\mathbf{d}_i = \frac{q_\psi(p_i)}{\|q_\psi(p_i)\|_2}, \qquad \mathbf{d}_u = \frac{q_\psi(p_u)}{\|q_\psi(p_u)\|_2}$$

The semantic similarity between an unseen procedure p_u and an existing procedure p_i is computed as

$$s_{ui} = \mathbf{d}_u^\top \mathbf{d}_i$$

For each existing procedure, let $\mathbf{e}_i$ denote its memory-token embedding and let $\mathbf{w}_i$ denote its corresponding TCRA weight. The similarity-based methods and the clustering-based method first compute coefficients a_{ui} over the existing procedures and then synthesize

$$\widetilde{\mathbf{e}}_u = \sum_{i=1}^{K} a_{ui}\mathbf{e}_i, \qquad \widetilde{\mathbf{w}}_u = \sum_{i=1}^{K} a_{ui}\mathbf{w}_i$$

The same coefficients are used for the memory embedding and TCRA parameters, so that all components of an unseen procedure are initialized consistently.

Similarity-based methods. We consider four methods that differ in which existing procedures are used and how their semantic similarities are converted into weights.

Top-3 Softmax (T3-SM). For each unseen procedure, we first select the three existing procedures with the highest semantic similarities:

$$\mathcal{N}_u = \mathrm{TopK}_{i\in\{1,\dots,K\}}(s_{ui}, 3)$$

We then apply a temperature-scaled softmax over these three similarities:

$$a_{ui} = \frac{\mathbb{I}[i \in \mathcal{N}_u] \exp(s_{ui}/\tau)}{\sum_{j\in\mathcal{N}_u} \exp(s_{uj}/\tau)}, \qquad \tau = 0.1$$

where $\mathbb{I}[\cdot]$ denotes the indicator function, which equals 1 when its argument is true and 0 otherwise. Thus, the memory embedding and routing parameters of a unseen procedure are synthesized primarily from the most similar existing procedure, while the other two selected procedures provide complementary information.

Top-3 Cosine (T3-Cos). Top-3 Cosine uses the same set $\mathcal{N}_u$, but directly normalizes the nonnegative cosine similarities instead of applying a softmax. Define

$$Z_u^{(3)} = \sum_{j\in\mathcal{N}_u} \max(s_{uj}, 0)$$

When $Z_u^{(3)} > 0$, the coefficients are

$$a_{ui} = \frac{\mathbb{I}[i \in \mathcal{N}_u] \max(s_{ui}, 0)}{Z_u^{(3)}}$$

If all three similarities are non-positive, the method assigns a uniform weight of $1/3$ to each selected procedure. Compared with Top-3 Softmax, this method preserves the relative cosine similarities without sharpening their differences through temperature scaling.

All Softmax (All-SM). All Softmax applies the temperature-scaled softmax to all K existing procedures:

$$a_{ui} = \frac{\exp(s_{ui}/\tau)}{\sum_{j=1}^{K} \exp(s_{uj}/\tau)}, \qquad \tau = 0.1.$$

This method does not explicitly restrict the number of contributing procedures. Consequently, every existing procedure receives a nonzero weight, although procedures with lower semantic similarity contribute less.

Positive Cosine (Pos-Cos). Positive Cosine synthesizes the new parameters from all existing procedures that have positive semantic similarity to the unseen procedure. Unlike the previous methods, it uses neither Top-k selection nor softmax normalization. Let the set of positively similar procedures be

$$\mathcal{S}_u^+ = \{i \mid s_{ui} > 0\}.$$

The positive cosine similarities are normalized directly:

$$a_{ui} = \frac{\mathbb{I}[i \in \mathcal{S}_u^+] s_{ui}}{\sum_{j\in\mathcal{S}_u^+} s_{uj}}.$$

Therefore, procedures with higher semantic similarity contribute more to the synthesized parameters, while procedures with non-positive similarity are excluded. If $\mathcal{S}_u^+$ is empty, the method uses the uniform average of all existing procedures, with $a_{ui} = 1/K$.

Clustering-based method: Clustering Mean (C-Mean). Clustering Mean first groups the existing procedure representations using cosine K-means. We use $C = 5$ clusters and at most 100 clustering iterations. Let $\boldsymbol{\mu}_c$ denote the normalized centroid of cluster c. The cluster selected for a unseen procedure is

$$c_u^* = \text{argmax}_{c\in\{1,\dots,C\}} \mathbf{d}_u^\top \boldsymbol{\mu}_c.$$

Let $\mathcal{C}_{c_u^*}$ denote the set of existing procedures assigned to the selected cluster. Nearest Mean assigns equal weights to all procedures in this cluster:

$$a_{ui} = \frac{\mathbb{I}[i \in \mathcal{C}_{c_u^*}]}{|\mathcal{C}_{c_u^*}|}.$$

Unlike the similarity-based methods, Nearest Mean uses the cluster structure of the existing procedure space rather than the individual similarity between p_u and each existing procedure.

Regression-based method: Ridge. Ridge fits a single global linear map from the semantic representation of a procedure to its learned parameters, instead of weighting individual donors. Let $\mathbf{D} \in \mathbb{R}^{K\times d}$ collect the existing representations and let $\boldsymbol{\Theta}^{(g)}$ collect their parameters of type g, where g ranges over the memory embedding, and the TCRA weight. The map is obtained by ridge regression, which adds an ℓ_2 penalty on the coefficients to the least-squares objective:

$$\widehat{\mathbf{B}}_g = \arg\min_{\mathbf{B}_g} \left\| \mathbf{D}\mathbf{B}_g - \boldsymbol{\Theta}^{(g)} \right\|_F^2 + \lambda \left\|\mathbf{B}_g\right\|_F^2, \qquad \lambda = 0.1.$$

The parameter of an unseen procedure is then predicted as $\widetilde{\boldsymbol{\theta}}_u^{(g)} = \mathbf{d}_u^\top \widehat{\mathbf{B}}_g$.

Compared with the other five methods, Ridge does not select or average a subset of existing procedures; it predicts an unseen procedure's parameters through a single global linear map.

Norm matching and model integration. Following the renormalization used by TokMem (Wu et al., 2026) to stabilize newly added memories, we rescale every synthesized parameter vector to the mean norm of the corresponding existing parameter bank. The target norms are

$$\rho_E = \frac{1}{K}\sum_{i=1}^{K} \|\mathbf{e}_i\|_2, \qquad \rho_W = \frac{1}{K}\sum_{i=1}^{K} \|\mathbf{w}_i\|_2.$$

The final parameters are

$$\mathbf{e}_u = \rho_E \frac{\widetilde{\mathbf{e}}_u}{\max(\|\widetilde{\mathbf{e}}_u\|_2, \epsilon)}, \qquad \mathbf{w}_u = \rho_W \frac{\widetilde{\mathbf{w}}_u}{\max(\|\widetilde{\mathbf{w}}_u\|_2, \epsilon)}$$

Each unseen procedure is synthesized independently using only the original K procedures. Its memory embedding and TCRA parameters are then appended to the trained model, while the backbone, existing memory embeddings, and existing TCRA parameters remain unchanged. Among the six methods, Top-3 Softmax provides the strongest overall cold-start performance in our comparison.

E Additional Experimental Results

E.1 Expanded Version of Results on Standard Setting

E.1.1 Detailed Results on More Backbones

Table 5 shows that TapMem improves the average Tool Selection and Argument Generation F1 over TokMem by 7.4% and 13.2% on Qwen 3.5 4B, and by 3.7% and 9.5% on Qwen 3.5 9B, respectively.

Table 7: Results of compositional recall with more tool calls on Llama 3.2 1B across 2-to-10 procedure calls. Δ denotes the improvement of TapMem over TokMem.

Calls	Method	Tool Selection(%)	Δ Tool Selection(%)	Argument(%)	Δ Argument(%)
2	TokMem	88.8	+2.5	75.9	+2.7
	Ours	**91.3**		**78.6**	
3	TokMem	89.4	+6.2	76.7	+6.8
	Ours	**95.6**		**83.5**	
4	TokMem	89.3	+2.3	72.3	+3.8
	Ours	**91.6**		**76.1**	
5	TokMem	88.9	+3.4	74.9	+3.9
	Ours	**92.3**		**78.8**	
6	TokMem	89.3	+3.6	75.4	+3.5
	Ours	**92.9**		**78.9**	
7	TokMem	87.6	+4.0	70.4	+5.1
	Ours	**91.6**		**75.5**	
8	TokMem	88.9	+4.2	72.2	+5.1
	Ours	**93.1**		**77.3**	
9	TokMem	84.9	+6.9	69.1	+6.5
	Ours	**91.8**		**75.6**	
10	TokMem	83.9	+7.6	65.1	+9.6
	Ours	**91.5**		**74.7**	
Avg.	TokMem	88.2	+4.3	73.0	+4.9
	Ours	**92.4**		**77.9**	

Table 8: Detailed ablation study on the effectiveness of ETS and TCRA in terms of F1.

ETS	TCRA	Tool Selection				Argument			
		2 calls	3 calls	4 calls	Avg.	2 calls	3 calls	4 calls	Avg.
Llama 3.2 1B									
		83.2	82.2	76.4	82.4	67.6	67.6	55.2	67.0
✓		83.6	85.3	75.8	84.0	68.0	70.7	54.8	68.6
✓	✓	88.8	89.6	81.0	**88.8**	74.9	74.1	65.0	**74.0**
Llama 3.2 3B									
		73.6	75.3	70.3	74.2	59.7	59.7	52.1	59.3
✓		78.3	81.1	77.9	79.6	62.4	66.5	57.9	64.2
✓	✓	81.1	82.7	81.4	**81.9**	65.5	65.4	58.3	**65.1**

Together with the results on the Llama backbones, these gains demonstrate that TapMem generalizes effectively across model families.

E.1.2 Detailed Results on Atomic Recall

Table 3 reports the results on the *Atomic Recall* setting, where TapMem improves Routing Accuracy over TokMem by 3.25%, 1.57% and 0.60% on the 0.5B, 3B, and 8B backbones, respectively, while slightly improving Rouge-L. The results show that TapMem also improves single-step procedure routing in this setting, and further supports the effectiveness of TapMem: beyond modeling transitions in compositional trajectories, its routing refinement also makes isolated memory-token selection more reliable. Table 4 compares TokMem and TapMem on the 20 categories with the largest number of samples. TapMem improves on most tasks, and the gains are relatively clear. On the two largest

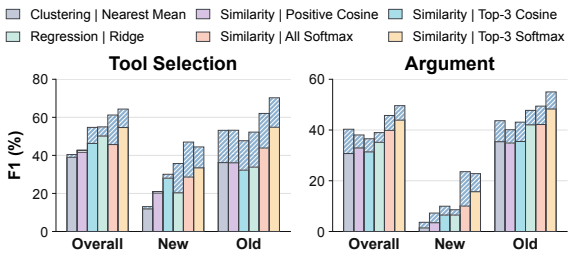


Figure 10: Comparison of cold-start methods. The hatched extension above each bar denotes the gain of TapMem over TokMem.

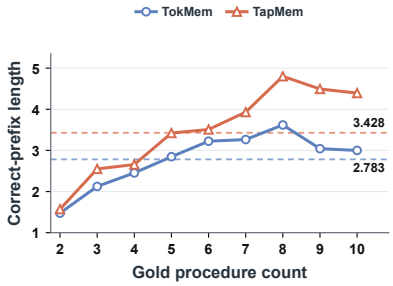


Figure 11: **Correct-prefix length** on Llama 3.2 1B.

categories, it improves Routing Accuracy by 5.19% on Question Answering and by 1.93% on Program Execution.

E.2 Expanded Version of Results of Long-Horizon Setting

E.2.1 Detailed Results on TaskBench and TRAJECT-Bench

Table 6 reports the detailed results on TaskBench and TRAJECT-Bench. On TaskBench, TapMem improves Tool Selection F1 and Argument Generation F1 over TokMem by 4.7% and 4.1% on average, respectively. On TRAJECT-Bench, the corresponding improvements are 5.0% and 5.7%. Together, these results demonstrate the effectiveness of TapMem on long-horizon procedure orchestration across both graph-structured dependencies and realistic multi-procedure trajectories.

E.2.2 Detailed Results on More Tool Calls

Table 7 reports the detailed compositional recall results with 2 to 10 procedure calls. TapMem consistently outperforms TokMem across all call counts, with average improvements of 4.3% in Tool Selection F1 and 4.9% in Argument Generation F1. The advantage becomes pronounced at longer horizons. At 9 calls, TapMem improves Tool Selection F1 and Argument Generation F1 by 6.9% and 6.5%, respectively, and the gains further increase to 7.6% and 9.6% at 10 calls. Meanwhile, from 2 to 10 calls, TokMem drops from 88.8% to 83.9% in Tool Selection F1 and from 75.9% to 65.1% in Argument Generation F1, whereas TapMem changes from 91.3% to 91.5% and from 78.6% to 74.7%, respectively. These results show that TapMem better maintains both procedure selection and argument generation as procedure chains grow longer, supporting the effectiveness of explicit transition modeling for long-horizon procedural orchestration.

E.3 Expanded Version of Results on Continual Setting

Figure 10 reports the cold-start results of the six synthesis methods on the unseen and the existing procedures. On Tool Selection, TapMem improves over TokMem for almost every synthesis method on both the unseen and the existing procedures, with gains of up to 18.4% on the unseen procedures and between 15.4% and 18.4% on the existing ones. On Argument F1, the advantage is consistent across all six methods, from 2.1% to 13.5% on the unseen procedures and from 5.2% to 8.3% on the existing ones. Under the best synthesis method, Top-3 Softmax, TapMem raises Tool Selection F1 from 33.5% to 44.4% and Argument F1 from 15.7% to 22.8% on procedures that are never seen during training. These results indicate that the benefit does not come from the cold-start strategy itself but from the transition mechanism that TapMem adds on top of tokenized procedural memory. This suggests that reliable procedure transitions remain valuable when procedural memory grows beyond the procedures seen during training.

Table 9: Gradient-flow ablation results.

Backbone	Gradient Design	Tool Selection	Argument
Llama 3.2 1B	Gradient Propagating	87.5	72.1
	Gradient Blocked	**89.7**	**74.3**
Llama 3.2 3B	Gradient Propagating	80.7	62.0
	Gradient Blocked	**81.4**	**62.7**
Llama 3.1 8B	Gradient Propagating	84.6	66.7
	Gradient Blocked	**86.7**	**74.7**

E.4 Expanded Version of Results on Ablation Study

Ablation study on components. Table 8 presents the detailed ablation results of different modules in our method. The results show that setting with ETS outperforms the unmodified TokMem baseline in most call-count settings. TCRA further improves the performance and achieves the best results.

Ablation study on gradient flow. Table 9 presents the results under two gradient designs across three backbones. Blocking gradients improves Tool Selection F1 by 2.2%, 0.7%, and 2.1%, and improves Argument F1 by 2.2%, 0.7%, and 8.0% on the 1B, 3B, and 8B models, respectively, demonstrating the effectiveness of our gradient design.

E.5 Analysis of Correct Prefix Length

Early routing decisions are critical in compositional procedure orchestration. An early mistake can cause subsequent procedure selections to follow an incorrect path. We measure this effect using consecutive correct-prefix length. It counts correct routing decisions from the start of a chain until the first incorrect or missing procedure.

As shown in Figure 11, TapMem improves the overall mean correct-prefix length from 2.78 to 3.43, and the per-length mean reaches 4.80. The advantage is concentrated on long chains: averaged over the last three procedure counts, the paired gain is 1.34, compared with 0.24 averaged over the first three, showing that TapMem's advantage in maintaining a correct routing prefix becomes increasingly clear as procedure chains grow longer.

Table 10: Trainable parameter counts in the compositional recall experiment.

Backbone	Fine-tuning	TokMem	Ours
Llama 3.2 1B	0.85M	0.10M	0.21M
Llama 3.2 3B	2.29M	0.15M	0.31M
Llama 3.1 8B	3.41M	0.20M	0.41M

F Comparison of Parameter Count

In Table 10, we report the number of trainable parameters for the parameterized methods across models of different scales. Our method requires substantially fewer trainable parameters than fine-tuning. Although it has more trainable parameters than TokMem, it still follows a linear growth trend with the embedding dimension.