UniMem: Complementary Episodic-to-Parametric Memory for Boundary-Agnostic Task Streams

Anonymous ACL submission

Abstract

Memory is essential for LLM agents to accumulate task experience and reuse task-specific execution strategies. However, real-world deployment over boundary-agnostic and evolving task streams exposes a fundamental stability-plasticity dilemma. External retrieval-based memory can rapidly absorb new evidence, but it often fails to internalize recurring execution patterns and incurs inference-time retrieval overhead. Parametric memory enables stable and efficient execution once learned, but typically relies on explicit task boundaries and fixed parameter budgets. Inspired by the human brain, which balances plasticity and stability through complementary episodic storage and gradual consolidation, we propose **UniMem**, a self-routing framework for autonomous memory management. UniMem uses learnable routing tokens as memory controllers, enabling adaptive coordination between complementary memory pathways: novel or sparse tasks are retained in an episodic buffer for retrieval-augmented execution, while recurring and reliable patterns are consolidated into expandable parametric memory. By decoupling task identification from task execution with routing tokens and parametric memory blocks, UniMem expands memory on demand without task labels during deployment or uncontrolled parameter growth. Experiments on long-horizon streaming task sequences show that UniMem consistently outperforms baselines while maintaining execution fidelity, achieving an average gain of 4.0 EM points across three backbone models.

1 Introduction

Memory is a core capability for LLM agents, enabling them to accumulate task experience and reuse task-specific execution strategies across interactions (Hu et al., 2026; Zhang et al., 2026; Zhao et al., 2024; Wu et al., 2026). Existing approaches to building memory broadly fall into two categories: external memory and parametric memory (Wu et al., 2025). External memory, represented by Retrieval-Augmented Generation (RAG) (Oche et al., 2025), stores instructions, demonstrations, or experiences outside model parameters and retrieves them during inference. Its main advantage is high plasticity: new evidence can be incorporated immediately without gradient updates, making it suitable for diverse and mixed tasks. In contrast, parametric memory internalizes reusable task knowledge into continuous trainable spaces. A widely adopted family of approaches is Parameter-Efficient Fine-Tuning (PEFT), including LoRA- and prompt-based variants (Wang et al., 2023; Li et al., 2024), which store task knowledge in lightweight modules while keeping the backbone frozen. Such parametric memory provides stable and efficient execution once the corresponding task pattern has been sufficiently learned.

However, practical deployment exposes a harder setting: LLM agents must operate over streaming tasks that are heterogeneous, boundary-agnostic, and continuously evolving (Delange et al., 2021; Yang et al., 2025; Bidaki et al., 2025). In this setting, relying on either external or parametric memory alone leads to a stability-plasticity dilemma. A purely external-memory solution remains flexible for mixed inputs, but it often shows limited generalization and suboptimal performance when task patterns recur, since retrieved examples or instructions do not reliably internalize reusable execution behavior into the model; it also incurs retrieval and context-processing overhead during inference. A purely parametric solution offers stable internalization, but it typically requires explicit task boundaries and a fixed parameter budget, making it difficult to accommodate heterogeneous streams whose task composition and scale change over time.

Complementary Learning Systems (CLS) theory provides a natural perspective for resolving this tension (McClelland et al., 1995; Kumaran et al., 2016). CLS argues that a single memory system

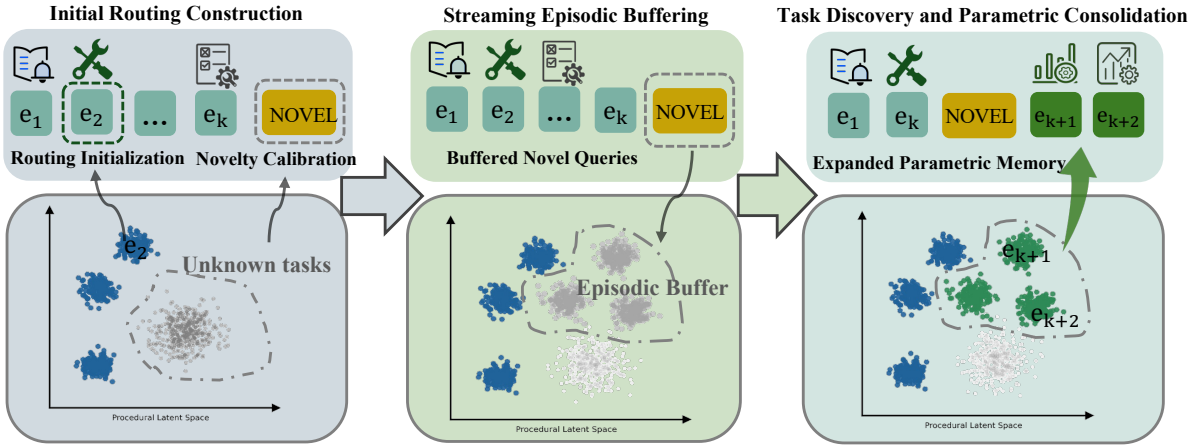

Figure 1: The episodic-to-parametric conceptual workflow of **UniMem**. **Initial Routing Construction** initializes known-task routing embeddings and calibrates the novelty sentinel $\mathbf{e}_{\text{NOVEL}}$. **Episodic Buffering** stores queries routed to $\mathbf{e}_{\text{NOVEL}}$ for retrieval-augmented execution. **Parametric Consolidation** clusters recurring buffered patterns and converts them into new routing embeddings with associated parametric memory units.

cannot simultaneously achieve high plasticity for rapidly absorbing new experiences and high stability for preserving reusable knowledge without interference. Instead, the human brain addresses this stability-plasticity trade-off through complementary memory systems: the hippocampus supports fast episodic storage of new experiences, while the neocortex gradually consolidates recurring structures into stable long-term knowledge (?Kumaran et al., 2016). This human-memory mechanism motivates our problem formulation: how can an LLM agent autonomously coordinate episodic retrieval with selective parametric consolidation in an boundary-agnostic task stream?

To address this challenge, we propose **UniMem**, a unified memory framework for open-world autonomous memory expansion. As shown in Figure 2, UniMem operationalizes the complementary-memory perspective through a self-routing episodic-to-parametric lifecycle. Its central mechanism is a set of learnable routing tokens that serve as memory controllers, learning task-discriminative signals to select an appropriate memory pathway for each query. Given an incoming query, UniMem compares it with consolidated task tokens and a calibrated novelty sentinel $\mathbf{e}_{\text{NOVEL}}$; this routing decision directly determines the subsequent memory access. Queries matched to known tokens activate the corresponding parametric memory block $\mathbf{P}_k$ for stable execution, whereas novel or uncertain queries are sent to an episodic buffer for retrieval-augmented support, avoiding premature parameter allocation. As evidence accumulates in the buffer, recurring patterns are clustered without task labels, filtered by a quality gate, and promoted into new memory units by allocating a fresh routing token paired with a parametric memory block. In this way, UniMem learns an adaptive memory management policy, retaining sparse evidence externally while autonomously consolidating reliable recurring patterns into expandable parametric memory.

In summary, our primary contributions are as follows:

- **Heterogeneous Streaming Tasks as a Memory Challenge:** We formulate heterogeneous, boundary-agnostic, and evolving task streams as a practical memory-management challenge for LLM-agent deployment, where relying solely on external or parametric memory exposes a stability-plasticity dilemma.
- **Human-Brain-Inspired Unified Memory Framework:** Inspired by Complementary Learning Systems theory, we propose UniMem, an automatic memory system that routes rare tasks to episodic memory and consolidates frequent patterns into parametric memory.
- **Extensive Empirical Validation:** Evaluations across diverse long-horizon streaming settings show that UniMem consistently outperforms existing baselines while maintaining robust task execution and memory stability during autonomous memory expansion.

2 Related Work

2.1 Memory-Augmented LLMs

Current memory augmentation paradigms broadly fall into two categories: external memory and parameter-based internal memory. External memory stores task-relevant knowledge outside model parameters and injects it through the context window as a dynamic workspace, with recent research exploring strategic knowledge extraction from agent experiences (Shinn et al., 2023; Zhao et al., 2024). To handle expanding external contexts over long horizons, adaptive management mechanisms like A-MEM (Xu et al., 2025) and MemAgent (Yu et al., 2025) have been introduced, while structural approaches organize historical contexts into logic-aware graphs for complex reasoning (Rasmussen et al., 2025; Zhang et al., 2025b; Anonymous, 2026). Although effective within their respective memory structures, these methods are typically structure-specific and lack an explicit mechanism for transforming episodic evidence into consolidated parametric memory.

Parameter-based adaptation methods reduce context overhead by storing task-specific behavior in lightweight trainable components while keeping most backbone parameters frozen. Prefix-Tuning (Li and Liang, 2021) optimizes continuous prefix vectors, LoRA and QLoRA (Hu et al., 2021; Dettmers et al., 2023) inject low-rank updates, and steering methods use small latent transformations to control model behavior (Han et al., 2024; Zhang et al., 2025a). Related work further explores instance-dependent prompt generation (Wu et al., 2022), ultra-compact prompt anchors (Liu et al., 2025), self-updatable memory pools (Wang et al., 2024), and reusable KV caches for reasoning (Gupta et al., 2026). However, these methods usually assume a predefined task set or a fixed adaptation structure. UniMem instead targets streaming settings by separating task identification from task execution through lightweight routing tokens and dedicated parametric memory blocks.

2.2 Parameterized Memory for Streaming Task Internalization

As LLM agents process streaming tasks, managing parametric memories without *catastrophic forgetting* remains a critical challenge (Bidaki et al., 2025; Chen et al., 2026). While multitask tuning inherently causes knowledge entanglement, standard modular techniques like LoRA suffer from representation interference when new task patterns are sequentially absorbed. To mitigate this parameter collision, various architectures explore different dimensions of parameter isolation to partition task-specific knowledge. One prominent direction focuses on macro-level module orchestration, which includes imposing orthogonal subspace constraints on independent adapters in O-LoRA (Wang et al., 2023) or employing dynamic routers for runtime adapter composition in L2R (Araujo et al., 2024). Concurrently, another parallel line of research targets micro-level parameter manipulation, either by deploying LoRA-driven Mixture-of-Experts (MoE) networks like MixLoRA (Li et al., 2024) to partition knowledge at the sub-task token level, or by encapsulating procedural memories into dedicated token slots as exemplified by Progressive Prompts (Razdaibiedina et al., 2023) and TOKMEM (Wu et al., 2026).

However, most existing methods assume a predefined task set and rely on manually allocated or fixed-capacity parameter structures, making them difficult to adapt to boundary-agnostic streaming scenarios. UniMem addresses it through a self-routing episodic-to-parametric lifecycle, where sparse queries are handled by retrieval and recurring tasks are consolidated into parametric memory.

3 Methodology

In this section, we formalize the proposed UniMem framework. We first establish the problem setting and the foundational token memory paradigm (Section 3.1). Then we introduce our self-routing memory architecture, which decouples lightweight routing tokens from high-capacity task-specific parametric memory blocks (Section 3.2). Finally, we detail the system's operational dynamics, including the two-stage episodic-to-parametric lifecycle from foundational memory initialization to autonomous task discovery and consolidation (Section 3.3).

3.1 Problem Setting and Preliminaries

Our framework builds upon the procedural token memory paradigm (Wu et al., 2026), which compresses task-specific execution procedures into single token embeddings alongside a frozen pretrained LLM, denoted as f_θ. For simplicity, we denote each reserved token by its trainable embedding $\mathbf{e}_k \in \mathbb{R}^d$, which represents the corresponding task procedure within the model. During training, the procedural token is inserted between

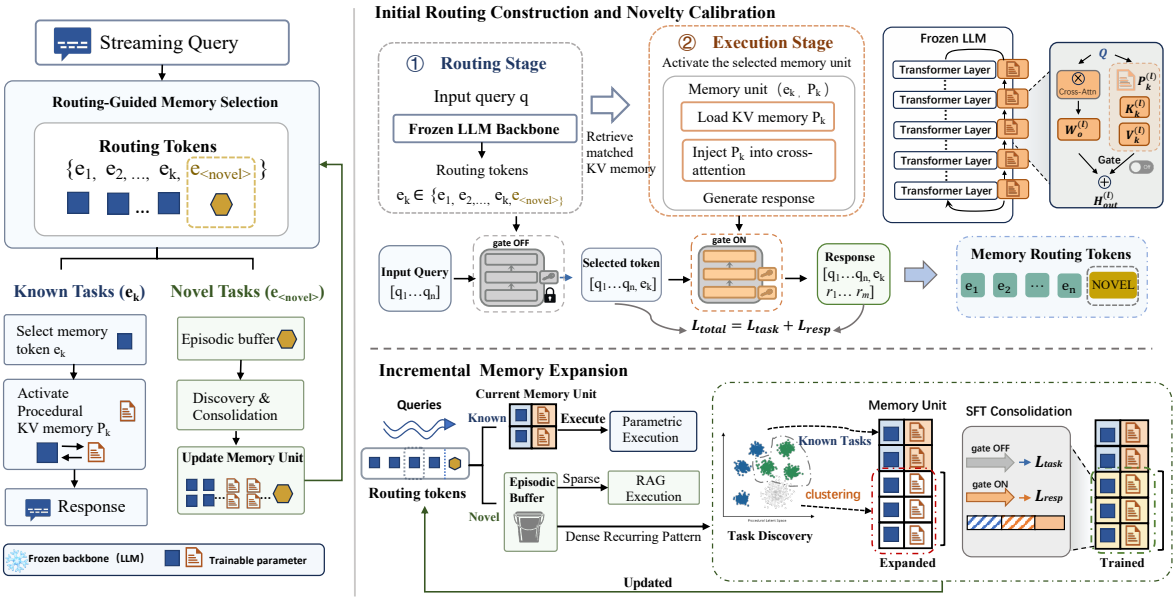

Figure 2: Architecture of UniMem. The left panel shows the overall self-routing architecture for assigning incoming queries to known-task memory units or the novelty pathway. On the right, the upper part illustrates routing-guided memory selection and execution, while the lower part shows episodic buffering and parametric consolidation during memory expansion.

the input query q and the target response r, producing a serialized procedure-response sequence: $x = (q_1, \ldots, q_{|q|},\ \mathbf{e}_k,\ r_1, \ldots, r_{|r|})$.

At inference, a query q is routed by computing a distribution over the memory embeddings based on the final hidden state $\mathbf{h}_q$:

$$P(\mathbf{e}_k \mid q) = \frac{\exp(\mathbf{h}_q^\top \mathbf{e}_k)}{\sum_{j=1}^{K} \exp(\mathbf{h}_q^\top \mathbf{e}_j)}. \tag{1}$$

While effective as a compact task identifier, a single token embedding is insufficient to encode complex procedural logic. We therefore retain the token embedding for routing and associate it with a separate high-capacity parametric memory block, as introduced in the next subsection.

3.2 Self-Routing Memory Architecture

UniMem uses a self-routing memory selection mechanism that separates task identification from task execution. Given an incoming query, this mechanism decides whether to activate an existing parametric memory unit or route the query into the novelty-driven expansion pathway.

Routing Tokens and Memory Selection. Extending the single-token memory formulation, UniMem retains routing tokens as lightweight selectors rather than treating them as complete memories. Specifically, UniMem maintains a routing token matrix $\mathbf{E} = [\mathbf{e}_1, \ldots, \mathbf{e}_K, \mathbf{e}_{\texttt{NOVEL}}]$, whose tokens fall into two functional classes:

- **Known Task Tokens** ($\mathbf{e}_k$): Each known task token corresponds to a consolidated task and is associated with a parametric memory unit $\mathbf{u}_k = (\mathbf{e}_k, \mathbf{P}_k)$, where $\mathbf{P}_k$ stores task-specific execution parameters.
- **Novelty Sentinel Token** ($\mathbf{e}_{\texttt{NOVEL}}$): The novelty sentinel token is calibrated to capture queries that do not match existing task representations and route them to the episodic buffer.

Token selection follows a confidence-based rule: UniMem activates the memory unit of the top known token only when its routing probability exceeds both the novelty-sentinel probability and a threshold τ_{route}; otherwise, the query enters the novelty pathway (Appendix A).

Known-Task Parametric Inference. Once a query is assigned to a known routing token $\mathbf{e}_k$, UniMem activates the associated memory unit $\mathbf{u}_k = (\mathbf{e}_k, \mathbf{P}_k)$ and enters the parametric execution phase. The memory block $\mathbf{P}_k$ is designed as a pluggable parameterization and can be instantiated with different PEFT-style modules. In this work, we instantiate $\mathbf{P}_k$ as a Procedural KV Memory composed of learnable, layer-wise key-value pairs $(\mathbf{K}_k^{(l)}, \mathbf{V}_k^{(l)})$.

During execution, the selected Procedural KV Memory contributes through the learned injection weights $\{g^{(l)}\}$. The resulting memory output is blended into the frozen self-attention output $\mathbf{H}^{(l)}_{\text{orig}}$:

$$\mathbf{H}^{(l)}_{\text{out}} = \mathbf{H}^{(l)}_{\text{orig}} + g^{(l)} \cdot \text{CrossAttn}\Big(\mathbf{Q}^{(l)}, \mathbf{K}^{(l)}_k, \mathbf{V}^{(l)}_k\Big), \tag{2}$$

where $\mathbf{Q}^{(l)}$ denotes the query states at layer l.

Decoupled Optimization Objectives. To align training with the retrieve-then-execute workflow, we optimize routing and execution with two objectives. The routing objective learns to select the appropriate routing token from the query alone, while the execution objective learns to generate the response using the selected parametric memory. Importantly, the execution loss is applied only to known-task samples, since samples routed to the novelty sentinel do not have an associated parametric memory block during calibration.

Let $z \in \{1, \ldots, K, \texttt{NOVEL}\}$ denote the discrete routing label of an input query, and let $\Theta_k = \{\mathbf{P}_k, \{g^{(l)}\}\}$ denote the task-specific parameters activated when a known task k is selected. The decomposed objective is:

$$\begin{aligned} \mathcal{L}_{\text{route}} &= -\log p(z \mid q_{1:n}; \mathbf{E}), \\ \mathcal{L}^{(k)}_{\text{exec}} &= -\frac{1}{m}\sum_{t=1}^{m} \log p(y_t \mid q_{1:n}, k, y_{<t}; \Theta_k). \end{aligned} \tag{3}$$

During routing, all parametric memory blocks are inactive, so the routing objective updates only the routing tokens. The execution loss $\mathcal{L}_{\text{exec}}$ is applied only for known tasks, updating the selected parametric memory block $\mathbf{P}_k$ and its gates.

3.3 Incremental Memory Expansion

UniMem starts with an initial set of known tasks and then expands memory from boundary-agnostic streaming samples through an episodic-to-parametric lifecycle. In Phase I, UniMem constructs the initial routing space and calibrates the novelty sentinel; in Phase II, it discovers recurring buffered patterns and consolidates qualified clusters into new parametric memory units.

Phase I: Routing Space Initialization. Phase I establishes a stable routing space for known tasks and calibrates a novelty sentinel for unseen queries. Given the initial known task set $\mathcal{T}^{(0)}$, UniMem first learns the routing tokens $\{\mathbf{e}_k\}_{k=1}^{K}$. After that, we compute the average routing norm $\bar{r} = \frac{1}{K}\sum_{k=1}^{K} \|\mathbf{e}_k\|$, which provides a fixed geometric scale for later memory expansion.

We then initialize the novelty sentinel $\mathbf{e}_{\texttt{NOVEL}}$ near the centroid of the known routing tokens. Specifically, a small Gaussian perturbation is added to the centroid direction, and the resulting vector is normalized to the same average norm:

$$\mathbf{e}_{\texttt{NOVEL}} = \bar{r} \cdot \frac{\bar{\mathbf{e}} + \boldsymbol{\epsilon}}{\|\bar{\mathbf{e}} + \boldsymbol{\epsilon}\|}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}\big(\mathbf{0}, (0.05\bar{r})^2\mathbf{I}\big), \tag{4}$$

where $\bar{\mathbf{e}}$ denotes the mean direction of the known routing tokens. This initialization places the novelty sentinel within the same routing space as known tasks, allowing it to compete with existing routing tokens rather than behave as an outlier.

To calibrate the novelty sentinel, we reserve a subset of tasks from the initial training pool and treat them as pseudo-unseen tasks. These held-out tasks are used only for novelty calibration and are not allocated known-task routing tokens. The routing layer is then trained with a $(K+1)$-way cross-entropy objective over the known routing labels and the novelty label `NOVEL`. Samples from observed known tasks are assigned to their corresponding routing labels, whereas samples from the reserved pseudo-unseen tasks are assigned to `NOVEL`. This calibration teaches the routing layer to separate known task queries from queries that should enter the episodic buffer.

Phase II: Autonomous Task Discovery and Parametric Consolidation. During streaming deployment, queries routed to the novelty sentinel are stored in the Episodic Buffer and temporarily handled by retrieval-augmented execution. When the buffer reaches capacity C, UniMem triggers unsupervised task discovery. To avoid false consolidation, it uses a conservative clustering pipeline: NCD (Cilibrasi and Vitanyi, 2004) measures task-pattern similarity, HDBSCAN (Campello et al., 2013) forms boundary-agnostic candidate clusters, and only clusters passing embedding-centroid outlier filtering and NCD-based cohesion checks are promoted. Details are provided in Appendix E.

For each qualifying cluster, UniMem dynamically allocates a fresh routing token $\mathbf{e}_{\text{new}}$ paired with a new parametric memory block $\mathbf{P}_{\text{new}}$, instantiated in this work as Procedural KV Memory. The new memory units are then jointly optimized on the clustered samples via supervised fine-tuning (SFT). Clusters that fail the quality gate remain in the Episodic Buffer and continue to be served by RAG-

Table 1: Generative execution performance on the SNI streaming task sequence scaling from 10 to 100 tasks. Results are averaged over three runs using the same seed-42 data split. Results are presented as **Exact Match (%) / ROUGE-L (%)**. The best results in each column are highlighted in **bold**, and the second-best results are underlined.

Method	Number of Tasks (T)			Avg.
	10	50	100	
Backbone: LLaMA-3.2-3B				
Base	29.60 / 47.71	32.64 / 45.97	31.80 / 45.72	31.35 / 46.47
RAG	37.00 / 58.43	37.22 / 51.79	35.21 / 49.45	36.48 / 53.22
LoRA	38.40 / 60.36	45.76 / 57.91	43.62 / 56.67	42.59 / 58.31
TokMem	38.40 / 61.80	41.06 / 54.67	40.54 / 55.07	40.00 / 57.18
UniMem (Ours)	**51.60 / 67.01**	**47.22 / 59.53**	**47.56 / 60.58**	**48.79 / 62.37**
Backbone: LLaMA-3.1-8B				
Base	39.80 / 58.09	39.40 / 53.33	39.28 / 53.20	39.49 / 54.87
RAG	45.20 / 64.83	42.61 / 59.12	41.56 / 56.65	43.12 / 60.20
LoRA	55.20 / 73.80	49.84 / 62.69	49.39 / 62.58	51.48 / 66.36
TokMem	56.20 / **74.08**	49.27 / 62.87	44.74 / 59.74	50.07 / 65.56
UniMem (Ours)	**58.00** / 73.75	**51.84 / 64.49**	**51.93 / 64.46**	**53.92 / 67.57**
Backbone: Qwen3-8B				
Base	40.80 / 53.10	45.84 / 57.28	44.36 / 57.72	43.67 / 56.03
RAG	41.80 / 60.35	42.86 / 58.53	42.26 / 58.68	42.31 / 59.19
LoRA	44.80 / 63.38	47.51 / 60.20	**50.44 / 63.48**	47.58 / 62.35
TokMem	39.80 / 59.67	44.45 / 57.46	44.35 / 57.92	42.87 / 58.35
UniMem (Ours)	**53.20 / 67.83**	**49.71 / 61.38**	49.94 / 62.22	**50.95 / 63.81**

based inference. This provides non-parametric coverage for sparse or long-tail tasks until sufficient evidence accumulates for future consolidation.

4 Experiments

In this section, we evaluate the effectiveness of UniMem on streaming task sequences and focus on whether UniMem can maintain execution performance under boundary-agnostic streams while supporting stable memory expansion.

4.1 Experimental Setup

Benchmarks and Streaming Protocols. We conduct evaluations on two representative benchmarks to assess UniMem across varying task complexities and scales. **(1) Super-Natural Instructions (SNI)** (Wang et al., 2022): To evaluate scalability across diverse tasks, we construct SNI streaming sequences with up to 100 tasks. We further simulate long-tail distributions by limiting a subset of tasks to only 10 training instances, causing them to remain episodic unless enough evidence accumulates for consolidation. Detailed construction procedures are provided in Appendix C. **(2) SuperGLUE Mixed Stream** (Wang et al., 2020): We select six complex reasoning tasks (BoolQ, CB, RTE, COPA, MultiRC, WiC) and blend them into a single, boundary-agnostic streaming task sequence.

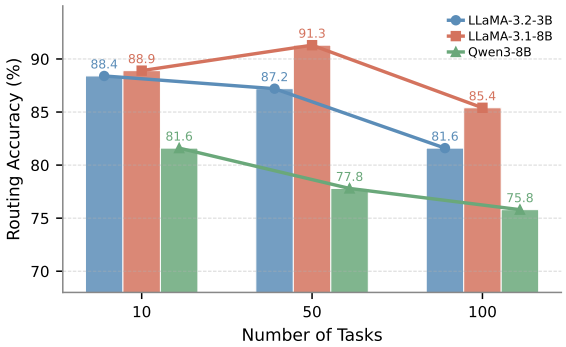

Figure 3: Routing token accuracy on SNI streaming task sequences under increasing task scales.

Implementation Details and Metrics. For SuperGLUE, we use LLaMA-1B and LLaMA-3B to evaluate reasoning streams under smaller backbone scales. For the larger and more diverse SNI generative stream, we further evaluate LLaMA-3.2-3B, LLaMA-3.1-8B, and Qwen3-8B to test scalability across stronger backbones. Performance on SuperGLUE is quantified via Average Accuracy ($\mathcal{A}$), while SNI performance is measured using both Exact Match (EM) and ROUGE-L.

Baselines. We compare UniMem with four representative baselines covering non-memory, external-memory, parametric-memory, and token-memory paradigms: **Base (Zero-Shot)**, the vanilla pre-

Table 2: Main results on the mixed SuperGLUE streaming task sequence. Performance is measured by final accuracy (%) after processing the full boundary-agnostic stream.

Method	BoolQ	CB	RTE	COPA	MultiRC	WiC	Avg. ($\mathcal{A}$)
Backbone: LLaMA-1B-Instruct							
Base	60.06	8.93	53.07	35.00	56.15	50.31	43.92
RAG	47.95	1.79	46.93	24.00	52.37	27.90	33.49
LoRA	76.97	51.79	68.23	68.00	72.73	51.72	64.91
TOKMEM	72.80	1.80	68.60	3.00	69.90	49.50	44.27
UniMem (Ours)	**81.96**	**69.64**	**80.14**	**78.00**	**75.91**	**57.68**	**73.89**
Backbone: LLaMA-3B-Instruct							
Base	81.19	60.71	72.92	39.00	70.77	50.31	62.48
RAG	75.87	78.57	74.37	81.00	68.03	52.66	71.75
LoRA	86.54	75.00	82.67	83.00	85.21	52.51	77.49
TOKMEM	84.40	64.30	80.90	**94.00**	84.00	66.30	78.98
UniMem (Ours)	**88.07**	**87.50**	**88.09**	92.00	**86.55**	**73.51**	**85.95**

trained model without task-specific adaptation; **Standard RAG**, which retrieves the k-nearest historical examples as in-context evidence; **Replay LoRA**, a rehearsal-based parametric baseline that interleaves a fixed proportion of previous samples (10%) into each training block; and **Adapted TOKMEM (Wu et al., 2026)**, a token-centric procedural memory baseline adapted to operate without explicit task labels in our streaming protocol. Detailed baseline implementation settings are provided in Appendix D.

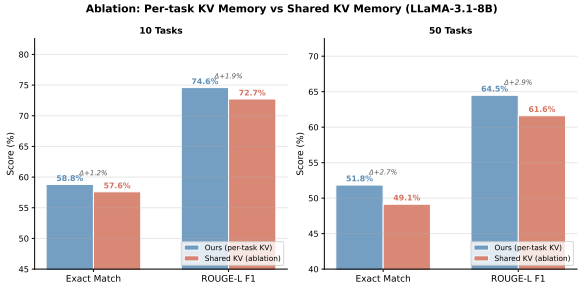

Figure 4: Ablation study of per-task vs. shared KV prefixes on LLaMA-3.1-8B.

4.2 Main Results and Analysis

Performance on SNI Streaming Tasks. Table 1 presents execution performance on the SNI streaming task sequence. UniMem achieves the best average scores across three backbone models, suggesting that UniMem can sustain strong task execution during long-term memory expansion.

The comparison with TOKMEM and Replay LoRA highlights the benefit of separating routing from execution. TOKMEM compresses task-specific behavior into a single token, whereas UniMem uses routing tokens only for task selection and stores execution behavior in separate parametric memory blocks. This design better supports diverse task procedures: on LLaMA-3.2-3B with 100 tasks, UniMem achieves 47.56% EM, compared with 40.54% for TOKMEM.

Replay LoRA is a strong parametric baseline, but its fixed parameter space can entangle heterogeneous task patterns. In contrast, UniMem consolidates recurring patterns into separate parametric memory units while keeping sparse tasks retrieval-based. As shown in Table 1, UniMem improves average EM over Replay LoRA by +6.20 and +2.44 points on LLaMA-3.2-3B and LLaMA-3.1-8B, respectively, with higher ROUGE-L scores.

We further evaluate routing token accuracy on the SNI streaming tasks. As shown in Figure 3, UniMem preserves reliable routing performance across task scales, suggesting that the routing layer can consistently identify the appropriate parametric memory unit. The LLaMA backbones achieve particularly strong accuracy, with LLaMA-3.1-8B remaining above 85% at Test-100. Although accuracy gradually decreases as semantically similar instructions and overlapping task patterns increase routing difficulty, the overall results indicate that UniMem maintains sufficiently discriminative routing tokens for long-horizon memory selection.

Performance on SuperGLUE Streams. Table 2 reports results on the boundary-agnostic and boundary-agnostic SuperGLUE Mixed Stream. UniMem achieves the highest average accuracy ($\mathcal{A}$) across both backbone scales, suggesting that it can handle implicit task transitions more effectively than baselines relying on a single shared adaptation space or token-centric memory representations.

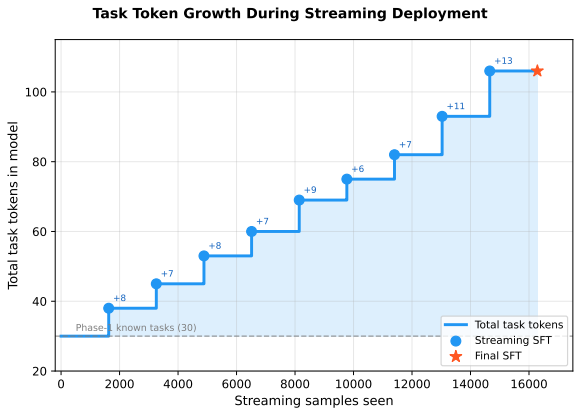

Figure 5: Cumulative memory growth during streaming deployment.

In particular, UniMem performs strongly on challenging tasks such as COPA and achieves the best average performance on the LLaMA-3B backbone.

Table 3: Ablation study on core components of **UniMem** across different streaming task scales. Results are reported as **EM (%) / ROUGE-L (%)**.

Method	Test-10		Test-50	
	EM	R-L	EM	R-L
Backbone: LLaMA-3.2-3B				
UniMem (full)	**51.60**	**67.01**	**47.22**	**59.53**
w/o RAG cache	—	—	46.45	58.97
w/o KV gate	35.80	56.43	31.22	45.33
Backbone: LLaMA-3.1-8B				
UniMem (full)	**58.00**	**73.75**	**51.84**	**64.49**
w/o RAG cache	—	—	50.90	63.10
w/o KV gate	37.40	57.10	30.16	42.05

4.3 Ablation and Qualitative Analysis

Effectiveness of Isolated Procedural KV Memory. We compare UniMem with a Shared KV Memory baseline on LLaMA-3.1-8B to assess the value of task-specific memory blocks. As shown in Figure 4, UniMem outperforms the shared variant in both 10-task and 50-task settings. The EM advantage increases from +1.2% to +2.7% as the task sequence grows, suggesting that shared parameters become more vulnerable to interference under greater task diversity. In contrast, task-specific KV memory helps preserve stable execution during streaming expansion.

Impact of Episodic Buffer Retrieval. The ablation of Episodic Buffer retrieval (w/o Episodic Buffer RAG) is omitted in the 10-task setting, where only one sparse query is observed and retrieval-augmented execution becomes highly volatile. In the 50-task setting, sparse and long-tail queries appear more frequently, and disabling Episodic Buffer retrieval leads to performance drop. This confirms that the Episodic Buffer provides useful non-parametric coverage for queries that do not yet justify parametric consolidation.

Criticality of the Gating Mechanism. Disabling the gating mechanism (w/o KV gate) causes substantial performance degradation across all settings. On the LLaMA-3.2-3B backbone, the EM score drops sharply in both the 10-task and 50-task settings, with a similar decline observed on LLaMA-3.1-8B. This confirms that learned gates are crucial for regulating the injection of task-specific Procedural KV Memory into the frozen backbone; without them, heterogeneous procedural signals interfere more strongly during execution.

Dynamics of Autonomous Task Discovery. Figure 5 tracks memory growth during streaming. Starting from 30 foundational tasks, UniMem allocates new parametric units only when buffered queries form recurring clusters. Across 16,000 samples covering 100 novel tasks, 76 new parametric units are created, while 10 sparse tasks remain in the episodic buffer due to insufficient evidence, and several semantically similar tasks merge into known units. This demonstrates that UniMem expands memory according to recurring task structure rather than allocating one unit per novel task.

5 Conclusion

In this paper, we introduced **UniMem**, a unified framework that bridges episodic retrieval and parametric consolidation for open-world memory expansion under streaming tasks. UniMem separates lightweight routing tokens from task-specific procedural memory, enabling LLM agents to handle heterogeneous task streams while keeping the backbone frozen. Through a novelty-guided lifecycle, sparse or uncertain queries are retained in an Episodic Buffer, whereas recurring workflows are selectively consolidated into persistent parametric memory. Experiments on long-horizon streaming task sequences show that UniMem maintains robust execution fidelity and outperforms token-centric memory methods and replay-based baselines. These results suggest a scalable direction for building self-evolving agents with expandable parametric memory.

Limitations

Although UniMem shows promising results for self-routing memory expansion under streaming tasks, two directions remain open. First, UniMem adopts a conservative consolidation strategy, keeping sparse or uncertain patterns in the Episodic Buffer until sufficient evidence is observed. While this avoids premature parameter allocation, more adaptive consolidation criteria may further improve memory utilization in dynamic streams.

Second, this work instantiates parametric memory with Procedural KV Memory and keeps the backbone frozen. Other parameter-efficient memory modules may offer different trade-offs in capacity, efficiency, and transferability, which we leave for future exploration.

References

Anonymous. 2026. Paper title. Under review.

Vladimir Araujo, Marie-Francine Moens, and Tinne Tuytelaars. 2024. Learning to route for dynamic adapter composition in continual learning with language models. *Preprint*, arXiv:2408.09053.

Seyed Amir Bidaki, Amir Mohammadkhah, Kiyan Rezaee, Faeze Hassani, Sadegh Eskandari, Maziar Salahi, and Mohammad M. Ghassemi. 2025. Online continual learning: A systematic literature review of approaches, challenges, and benchmarks. *Preprint*, arXiv:2501.04897.

Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. 2013. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*, pages 160–172. Springer.

Hongyang Chen, Zhongwu Sun, Hongfei Ye, Kunchi Li, and Xuemin Lin. 2026. Continual learning in large language models: Methods, challenges, and opportunities. *Preprint*, arXiv:2603.12658.

Rudi Cilibrasi and Paul Vitanyi. 2004. Clustering by compression. *Preprint*, arXiv:cs/0312044.

Matthias Delange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Ales Leonardis, Greg Slabaugh, and Tinne Tuytelaars. 2021. A continual learning survey: Defying forgetting in classification tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, page 1–1.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *Preprint*, arXiv:2305.14314.

Sharut Gupta, Phillip Isola, Stefanie Jegelka, David Lopez-Paz, Kartik Ahuja, Mark Ibrahim, and Mohammad Pezeshki. 2026. Reasoncache: Teaching llms to reason without weight updates. *Preprint*, arXiv:2602.02366.

Chi Han, Jialiang Xu, Manling Li, Yi Fung, Chenkai Sun, Nan Jiang, Tarek Abdelzaher, and Heng Ji. 2024. Word embeddings are steers for language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16410–16430.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *Preprint*, arXiv:2106.09685.

Yuyang Hu, Shichun Liu, Yanwei Yue, Guibin Zhang, Boyang Liu, Fangyi Zhu, Jiahang Lin, Honglin Guo, Shihan Dou, Zhiheng Xi, Senjie Jin, Jiejun Tan, Yanbin Yin, Jiongnan Liu, Zeyu Zhang, Zhongxiang Sun, Yutao Zhu, Hao Sun, Boci Peng, and 28 others. 2026. Memory in the age of ai agents. *Preprint*, arXiv:2512.13564.

Dharshan Kumaran, Demis Hassabis, and James L McClelland. 2016. What learning systems do intelligent agents need? complementary learning systems theory updated. *Trends in cognitive sciences*, 20(7):512–534.

Dengchun Li, Yingzi Ma, Naizheng Wang, Zhengmao Ye, Zhiyuan Cheng, Yinghao Tang, Yan Zhang, Lei Duan, Jie Zuo, Cal Yang, and Mingjie Tang. 2024. Mixlora: Enhancing large language models fine-tuning with lora-based mixture of experts. *Preprint*, arXiv:2404.15159.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *Preprint*, arXiv:2101.00190.

Yiyang Liu, James C. Liang, Heng Fan, Wenhao Yang, Yiming Cui, Xiaotian Han, Lifu Huang, Dongfang Liu, Qifan Wang, and Cheng Han. 2025. All you need is one: Capsule prompt tuning with a single vector. *Preprint*, arXiv:2510.16670.

James L. McClelland, Bruce L. McNaughton, and Randall C. O'Reilly. 1995. Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. *Psychological review*, 102 3:419–457.

Agada Joseph Oche, Ademola Glory Folashade, Tirthankar Ghosal, and Arpan Biswas. 2025. A systematic review of key retrieval-augmented generation (rag) systems: Progress, gaps, and future directions. *Preprint*, arXiv:2507.18910.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. 2025. Zep: A temporal knowledge graph architecture for agent memory. *Preprint*, arXiv:2501.13956.

Anastasia Razdaibiedina, Yuning Mao, Rui Hou, Madian Khabsa, Mike Lewis, and Amjad Almahairi. 2023. Progressive prompts: Continual learning for language models. *Preprint*, arXiv:2301.12314.

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

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2020. Superglue: A stickier benchmark for general-purpose language understanding systems. *Preprint*, arXiv:1905.00537.

Xiao Wang, Tianze Chen, Qiming Ge, Han Xia, Rong Bao, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. 2023. Orthogonal subspace learning for language model continual learning. *Preprint*, arXiv:2310.14152.

Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Anjana Arunkumar, Arjun Ashok, Arut Selvan Dhanasekaran, Atharva Naik, David Stap, Eshaan Pathak, Giannis Karamanolakis, Haizhi Gary Lai, Ishan Purohit, Ishani Mondal, Jacob Anderson, Kirby Kuznia, Krima Doshi, Maitreya Patel, and 21 others. 2022. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. *Preprint*, arXiv:2204.07705.

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. 2024. Memoryllm: Towards self-updatable large language models. *Preprint*, arXiv:2402.04624.

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

Zhuofeng Wu, Sinong Wang, Jiatao Gu, Rui Hou, Yuxiao Dong, V. G. Vinod Vydiswaran, and Hao Ma. 2022. Idpg: An instance-dependent prompt generation method. *Preprint*, arXiv:2204.04497.

Zijun Wu, Yongchang Hao, and Lili Mou. 2026. Tokmem: One-token procedural memory for large language models. *Preprint*, arXiv:2510.00444.

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

Qinwen Yang, Liyuan Wang, Joerg Wicker, and Gillian Dobbie. 2025. Continual learning: A systematic literature review. *Neural Networks*, page 108226.

Hongli Yu, Tinghong Chen, Jiangtao Feng, Jiangjie Chen, Weinan Dai, Qiying Yu, Ya-Qin Zhang, Wei-Ying Ma, Jingjing Liu, Mingxuan Wang, and Hao Zhou. 2025. Memagent: Reshaping long-context llm with multi-conv rl-based memory agent. *Preprint*, arXiv:2507.02259.

Chenheng Zhang, Tianqi Du, Jizhe Zhang, Mingqing Xiao, Yifei Wang, Yisen Wang, and Zhouchen Lin. 2025a. Language ranker: A lightweight ranking framework for llm decoding. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Chenheng Zhang, Yijun Lu, Lizhe Fang, Chunyuan Zheng, Jiajun Chai, Xiaohan Wang, Guojun Yin, Wei Lin, Yisen Wang, and Zhouchen Lin. 2026. Rethinking personalization in large language models at the token level. In *Proceedings of the 43nd International Conference on Machine Learning*.

Guibin Zhang, Muxin Fu, Guancheng Wan, Miao Yu, Kun Wang, and Shuicheng Yan. 2025b. G-memory: Tracing hierarchical memory for multi-agent systems. *Preprint*, arXiv:2506.07398.

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

A Routing Selection Mechanism

This appendix describes the inference-time routing rule used by UniMem. The rule is designed to support two complementary novelty signals: explicit novelty recognition by the learned novelty token, and low-confidence rejection when no known task can confidently explain the input.

Given an input query $\mathbf{x}$, the frozen backbone produces the last-token hidden state $\mathbf{h} \in \mathbb{R}^d$. UniMem then computes a routing distribution over $T+1$ candidates, consisting of T known-task routing tokens $\{e_1, \ldots, e_T\}$ and one novelty sentinel e_{NOV}. We index the novelty sentinel by $i = 0$:

$$p_i = \frac{\exp(\mathbf{h}^\top e_i/\sqrt{d})}{\sum_{j=0}^{T} \exp(\mathbf{h}^\top e_j/\sqrt{d})}, \quad i \in \{0, 1, \ldots, T\}. \tag{5}$$

Let p_0 denote the probability assigned to the novelty sentinel. We define the strongest known-task candidate as

$$i^\star = \arg\max_{i \in \{1,\ldots,T\}} p_i, \quad p^\star = p_{i^\star}. \tag{6}$$

Here, $i^\star$ is the best known-task routing token and $p^\star$ is its confidence.

UniMem activates a known parametric memory unit only when two conditions are simultaneously satisfied:

$$p^\star > p_0 \quad \text{and} \quad (\, p^\star \geq \tau_{\mathrm{route}}). \tag{7}$$

Otherwise, the query is routed to the novelty pathway. Equivalently, the final routing decision is

$$\hat{y} = \begin{cases} i^\star, & \text{if } p^\star > p_0 \text{ and } (p^\star \geq \tau_{\text{route}}), \\ e_{\text{NOV}}, & \text{otherwise.} \end{cases} \tag{8}$$

This rule captures two ways in which a query can be treated as novel. First, if the novelty sentinel itself receives higher probability than every known-task token, the query is explicitly recognized as novel. Second, even if a known-task token has the largest probability among known tasks, UniMem rejects the known memory activation when the confidence $p^\star$ is below the threshold τ_{route}. This prevents low-confidence queries from activating an incorrect Procedural KV Memory block.

When $\tau_{\text{route}} = 0$, the rule reduces to sentinel-based routing between the strongest known-task token and the novelty sentinel. When $\tau_{\text{route}} > 0$, the threshold introduces an additional conservative rejection mechanism. In our experiments, τ_{route} is set as 0.7.

We use this max-known-probability gate instead of entropy or margin. Entropy can be sensitive to the number of weak competing routing tokens, while margin-based rejection introduces an additional criterion between the top two candidates. The scalar $p^\star$ directly measures whether any known memory unit claims the query with sufficient confidence, making the novelty decision simple and interpretable.

B Procedural KV Memory: Implementation Details

Cross-Attention Mechanics. The Procedural KV Memory in Eq. 2 is implemented as a set of task-specific key-value slots at each Transformer layer. For a selected known task k, the current hidden states at layer l are projected into query states $\mathbf{Q}^{(l)}$ using the frozen query projection $\mathbf{W}_Q^{(l)}$. We do not apply rotary position encoding (RoPE) to the Procedural KV Memory, since these key-value slots encode task-level procedural semantics rather than position-dependent token interactions. No causal mask is applied to this cross-attention, allowing the selected procedural memory to be globally accessible to every sequence position.

When the number of memory slots is p=1, the cross-attention reduces to a simple and interpretable form. Since there is only one key, the attention weight over the memory slot is always 1, and every position receives the same task-specific value vector after output projection. In this case, the Procedural KV Memory behaves as a layer-wise task-specific residual signal added to the frozen backbone representation.

Grouped-Query Attention (GQA) Handling. For models using grouped-query attention (GQA), such as Llama-3.2-3B with H_q=32 query heads and H_{kv}=8 key-value heads, the Procedural KV Memory is parameterized over H_{kv} heads to reduce parameter cost. Before computing cross-attention, the stored key-value pairs are repeated H_q/H_{kv} times to match the number of query heads. For Llama-3.2-3B, this corresponds to a repetition factor of 4.

Gate Parameterization. Each layer is equipped with a learnable scalar gate that controls the contribution of the selected Procedural KV Memory. We use a clamped parameterization

$$g^{(l)} = \text{clamp}(\tilde{g}^{(l)}, 0, g_{\max}), \tag{9}$$

rather than a sigmoid gate. This choice provides a directly interpretable injection strength and avoids vanishing gradients near sigmoid saturation. We initialize $\tilde{g}^{(l)}$=0.01, so that the model starts close to the frozen backbone behavior and the Procedural KV Memory contributes only weakly at the beginning of training.

Parameter Budget. For a model with L layers, H_{kv} key-value heads, head dimension d_h, hidden dimension d, and p memory slots per layer, each task adds trainable parameters.

$$\underbrace{d}_{\text{routing token}} + \underbrace{2LH_{\text{kv}}d_h p}_{\text{Procedural KV Memory}} + \underbrace{L}_{\text{gates}} \tag{10}$$

For Llama-3.2-3B (L=28, H_{kv}=8, d_h=128, d=3072, p=1), this yields

$$3{,}072 + 57{,}344 + 28 = 60{,}444$$

parameters per task, which is several orders of magnitude smaller than the frozen 3B backbone.

Training Hyperparameters We summarize the main training and system hyperparameters used in the SNI streaming experiments.

The Procedural KV Memory uses a larger learning rate of 5×10^{-3}, while the routing tokens are trained with a learning rate of 1×10^{-3}.

Hyperparameter	Value
Batch size	2
Gradient accumulation	2
Effective batch size	4
Maximum sequence length	2560
Precision	bfloat16
Epochs per training round	1
Routing-token learning rate	1×10^{-3}
Procedural KV Memory learning rate	5×10^{-3}

Table 4: Training hyperparameters shared across the SNI streaming experiments.

C Streaming Protocol Details

C.1 Artifact use.

We use Super-NaturalInstructions and SuperGLUE as public benchmark datasets for research evaluation. We follow the original licenses and terms of use of all datasets and pretrained models used in this work, including the LLaMA and Qwen model families. We do not redistribute the original datasets or pretrained model weights. The constructed streaming splits are used only for controlled research evaluation.

C.2 Task Pool Construction

We construct the streaming task pool from the full English task set of Natural Instructions 2.8. To ensure reproducibility, tasks are first sorted lexicographically and then shuffled with a fixed random seed, which is set to 42 by default. From the shuffled list, we select N_{total} tasks and partition them into four disjoint subsets: T_{stream}, T_{known}, T_{calib}, and T_{held}, corresponding to the unknown task stream, initial known tasks, novelty-calibration tasks, and unused held-out tasks, respectively.

The sizes of the first three subsets are N_{stream}, N_{known}, and N_{calib}. Here, N_{known}, N_{calib}, and N_{stream} denote the number of initial known tasks, novelty-calibration tasks, and unknown streaming tasks, respectively. In our experiments, we set $N_{\text{known}} = 20$ and $N_{\text{calib}} = 20$ for routing-space initialization and novelty calibration. The unknown streaming tasks are kept strictly disjoint from the novelty-calibration tasks and unused held-out tasks to avoid label conflicts. In particular, a streaming task should not appear as a novelty-calibration task, otherwise it would be treated both as a novelty target during calibration and later as a specific discovered cluster during streaming expansion.

C.3 Per-Task Sample Split

For each unknown streaming task, samples are sorted by a deterministic hash

$$h(x) = \text{MD5}(\text{id} \,||\, \text{input} \,||\, \text{output}),$$

where $||$ denotes string concatenation. This deterministic ordering ensures that the same examples are selected across different runs. The sorted samples are then split into test, train, and validation subsets:

$$\underbrace{x_{1:N_{\text{test}}}}_{\text{test}} \quad \underbrace{x_{N_{\text{test}}+1:N_{\text{test}}+N_{\text{train}}}}_{\text{train}} \quad \underbrace{x_{N_{\text{test}}+N_{\text{train}}+1:\cdots}}_{\text{validation}}.$$

Test samples are always drawn from the hash-sorted head, so the evaluation set remains fixed regardless of how many training samples are used. Unless otherwise specified, we use $N_{\text{train}} = 200$ and $N_{\text{test}} = 50$ per task.

C.4 Sparse Task Simulation

To simulate long-tail task distributions in open-world environments, a fraction r_{sparse} of unknown streaming tasks is randomly designated as sparse tasks using an independent random generator with seed $42 + 9999$. By default, $r_{\text{sparse}} = 10\%$. Each sparse task is allocated only $N_{\text{sparse}} = 10$ training samples, which is far below the minimum cluster size $S_{\min} = 50$ used by HDBSCAN. As a result, sparse tasks cannot form valid clusters for parametric consolidation and are instead served through the episodic RAG fallback during inference. This design tests whether UniMem can avoid premature parameter allocation for insufficiently supported task patterns.

C.5 Stream Ordering

Training samples from unknown streaming tasks are presented to the model as an boundary-agnostic stream. We support two stream ordering modes in the implementation.

Global shuffle. When this option is set to 0, all training samples from unknown streaming tasks are globally shuffled. This is the default setting used in our experiments. It simulates a mixed deployment stream in which task identity is unknown and samples from different tasks arrive interleaved.

Group-wise stream. When this option is set to G, tasks are divided into consecutive groups of size G. Within each group, samples from all G tasks are pooled and locally shuffled, and groups arrive sequentially. This setting preserves temporal

locality among related tasks while still hiding task labels within each group.

In the main SNI experiments, we use the group-wise stream setting to simulate a multi-task deployment scenario where task groups arrive sequentially over time. Within each group, samples from multiple tasks are mixed without task labels or explicit task boundaries, requiring the model to perform routing and memory expansion under local task heterogeneity. For SuperGLUE, since the task set is relatively small, we instead use the global-shuffle setting, where samples from all tasks are fully mixed into a single boundary-agnostic stream.

D Baseline Implementation Details

This appendix provides the implementation details of the baselines used in our experiments. For a controlled comparison, all trainable baselines are constructed from the same streaming training data and evaluated on the same held-out test split. Unless otherwise specified, all methods use greedy decoding during inference.

D.1 Zero-Shot Baseline

The zero-shot baseline directly applies the frozen pretrained model without any task-specific adaptation. In our experiments, we use the same backbone models as in the main evaluation.

At inference time, each test sample from the task set is formatted as a standard chat prompt, where the task instruction is prepended to the input query. The model then generates the response with greedy decoding, i.e., `do_sample=False`. No task-specific parameters are introduced or updated. This baseline measures the inherent instruction-following and task generalization ability of the frozen pretrained model.

D.2 Retrieval-Augmented Generation Baseline

The retrieval-augmented generation (RAG) baseline augments zero-shot inference with retrieved in-context demonstrations. Its retrieval corpus is constructed from the training set, which is the same data observed by the streaming memory baselines. This ensures that RAG and the trainable streaming methods have access to the same pool of task evidence.

For each test query, we form the retrieval query by concatenating the task instruction and the input query. We then encode the retrieval query and all candidate training samples with Qwen3-Embedding-0.6B. Similarity is computed by cosine similarity over ℓ_2-normalized embeddings. The top-k most similar training samples are retrieved, where we set $k = 3$ in all experiments.

The retrieved samples are formatted as few-shot demonstrations and prepended to the input prompt before generation. To control the context length, we impose a character budget of 4,000 characters for the retrieved demonstrations. The final prompt is decoded greedily without any gradient update. Thus, this baseline evaluates whether non-parametric retrieval alone can provide sufficient task adaptation under the same streaming data source.

D.3 LoRA Fine-Tuning Baseline

The LoRA baseline applies parameter-efficient fine-tuning to the training data. The pretrained backbone remains frozen, and only the LoRA adapter parameters are updated. The training input is formatted with the same chat template used by the memory-based methods, where the task instruction is concatenated into the user turn. The loss is computed only on the assistant response tokens, while the instruction and query tokens are masked with the ignore label.

The LoRA configuration is shown in Table 5.

Hyperparameter	**Value**
LoRA rank r	8
LoRA α	32
Dropout	0.1
Target modules	`q_proj`, `v_proj`
Learning rate	5×10^{-5}
Batch size	1
Gradient accumulation	4
Epochs	1
Maximum sequence length	2560
Precision	bfloat16

Table 5: Hyperparameter configuration of the LoRA fine-tuning baseline.

To ensure data alignment, the LoRA baseline uses the same data preparation procedure as the streaming memory methods. Specifically, it calls the same `prepare_incremental_data` routine with identical arguments, including the random seed, the number of tasks, and the per-task training size. Therefore, LoRA observes exactly the

same training samples and is evaluated on the same held-out test set.

D.4 LoRA with Continual Replay

We further evaluate a continual replay variant of LoRA to mitigate catastrophic forgetting under sequential training. This baseline follows the same LoRA configuration as described above, but introduces block-wise cumulative replay during adaptation.

The tasks are divided into consecutive blocks of $B = 10$ tasks. When training on the first block, no replay is used because no previous samples are available. For each subsequent block $b > 0$, a fraction $\rho = 0.1$ of the current block's training samples is replaced by samples uniformly drawn without replacement from the cumulative pool of all previous blocks. The number of replay samples is computed as

$$n_{\text{replay}} = \lfloor n_{\text{current}} \times \rho \rfloor , \tag{11}$$

where n_{current} denotes the number of samples in the current training block.

Replay samples are interleaved into the current training sequence at approximately uniform intervals. Specifically, one replay sample is inserted every $\left\lfloor \frac{n_{\text{current}}}{n_{\text{replay}}+1} \right\rfloor$ training steps. After training on block b, the samples from this block are added to the cumulative replay pool, allowing later blocks to replay from all earlier tasks.

We preserve the block-wise training order and do not globally shuffle the stream, so that temporal task locality is maintained. All other training hyperparameters are identical to the plain LoRA baseline. In the main experiments, this replay-enhanced variant is reported as the Replay LoRA baseline.

D.5 Adapted TOKMEM Baseline

We include an adapted TOKMEM baseline to provide a direct architectural comparison with UniMem. TOKMEM represents each task procedure with a single trainable memory token. In our implementation, we adapt TOKMEM to the same streaming protocol as UniMem, so that the two methods differ only in whether a routed memory token is paired with an additional high-capacity parametric memory block.

Specifically, the adapted TOKMEM baseline uses the same routing-space initialization, novelty-sentinel calibration, episodic buffering, task discovery, and consolidation schedule as UniMem. For each consolidated task or discovered cluster, TOKMEM allocates a trainable procedural token e_k. During inference, the routing rule selects the most appropriate token in the same way as UniMem. If the query is assigned to a known task token, the selected token e_k is inserted into the input sequence and used directly as the task-specific procedural memory. If the query is routed to the novelty sentinel, it is handled by the same episodic retrieval fallback as in UniMem.

The key difference is that TOKMEM does not introduce an additional Procedural KV Memory block. In other words, each memory unit in TOKMEM contains only

$$u_k^{\text{TOKMEM}} = \{e_k\}, \tag{12}$$

whereas each UniMem memory unit contains both a routing token and an execution memory block:

$$u_k^{\text{UniMem}} = \{e_k, P_k\}. \tag{13}$$

Therefore, the selected token in TOKMEM must simultaneously serve as the task identifier and the execution memory. By contrast, UniMem uses e_k only as a lightweight selector and stores task-specific execution behavior in the associated Procedural KV Memory P_k.

Training follows the same response-supervised objective and data split as UniMem. For known or consolidated tasks, the loss is computed only on the assistant response tokens, while the instruction, query, and memory-token positions are masked. Only the memory token parameters are updated for TOKMEM, and the frozen backbone remains unchanged. No layer-wise key-value slots, cross-attention injection, or memory gates are used in this baseline.

This adapted TOKMEM baseline isolates the effect of decoupling routing from execution. If a single procedural token is sufficient to encode the task-specific behavior, TOKMEM should perform similarly to UniMem. The observed performance gap therefore reflects the benefit of augmenting lightweight routing tokens with dedicated Procedural KV Memory for high-capacity execution.

E NCD-Based Streaming Clustering

This appendix provides the implementation details of the unsupervised task discovery module. The goal of this module is to identify recurring latent task patterns from boundary-agnostic samples

routed to the novelty sentinel. Since no task labels or explicit task boundaries are available during streaming deployment, UniMem performs clustering directly over the episodic cache and promotes only high-quality clusters into parametric procedural memory.

E.1 Overview

Let $\mathcal{B}$ denote the episodic cache that stores samples routed to the novelty sentinel e_{NOVEL}. Incoming samples are appended to $\mathcal{B}$ until the cache size reaches a predefined capacity C, where we set $C = 1600$ by default. Once $|\mathcal{B}| \geq C$, UniMem triggers one round of structural discovery.

Each clustering round consists of three stages:

1. computing sample-level procedural distances with Normalized Compression Distance (NCD);
2. applying HDBSCAN to discover density-based candidate clusters without task labels;
3. filtering noisy members and low-quality clusters before using the remaining clusters for procedural consolidation.

Only clusters that pass all filtering and quality checks are used to initialize new routing tokens and Procedural KV Memory blocks. Noise points, unassigned samples, and rejected clusters are kept in the episodic cache so that sparse tasks can continue accumulating evidence over future streaming rounds.

E.2 Text Representation

For each boundary-agnostic sample, we construct a text representation that captures both task-level and instance-level information. Let I_i denote the task instruction and q_i denote the input query of the i-th sample. We form the clustering text x_i by concatenating a truncated instruction prefix and a truncated query:

$$x_i = \text{Trunc}_{\lfloor L/2 \rfloor}(I_i) \mid\mid \text{Trunc}_{L-\lfloor L/2 \rfloor}(q_i), \quad (14)$$

where $||$ denotes string concatenation and $L = 500$ is the maximum number of characters. This representation allocates approximately half of the input budget to the instruction and half to the query, ensuring that the clustering signal reflects both the abstract task procedure and the concrete input pattern. All NCD computations are performed over the resulting character-level text.

E.3 NCD Distance Computation

We use Normalized Compression Distance (NCD) to measure procedural similarity between two samples. Given two clustering texts x_i and x_j, NCD is defined as

$$\text{NCD}(x_i, x_j) = \frac{C(x_i x_j) - \min\{C(x_i), C(x_j)\}}{\max\{C(x_i), C(x_j)\}}, \quad (15)$$

where $C(\cdot)$ denotes the compressed byte length computed by `gzip`, and $x_i x_j$ denotes the concatenation of the two texts. Smaller NCD values indicate higher similarity, while larger values indicate lower similarity. In contrast to embedding-based distances, NCD does not require a learned representation and is sensitive to repeated lexical, syntactic, and structural patterns that often characterize task-specific procedures.

We use two computation modes depending on the cache size $n = |\mathcal{B}|$.

Full pairwise mode. When $n \leq 2000$, we compute the full pairwise NCD matrix $D \in \mathbb{R}^{n \times n}$:

$$D_{ij} = \text{NCD}(x_i, x_j). \quad (16)$$

This mode preserves the exact pairwise distance structure among all cached samples and is used as the default mode for most clustering rounds. Its computational complexity is $O(n^2)$ compression calls.

Anchor mode. When $n > 2000$, computing the full pairwise matrix becomes expensive. We therefore switch to an anchor-based approximation. Specifically, we randomly select $K = 100$ anchor samples $\{a_k\}_{k=1}^{K}$ from the current cache. Each sample x_i is represented by a K-dimensional NCD feature vector $\mathbf{f}_i \in \mathbb{R}^K$:

$$f_{i,k} = \text{NCD}(x_i, a_k). \quad (17)$$

This reduces the number of compression calls from $O(n^2)$ to $O(nK)$. The resulting feature matrix is then projected to 30 dimensions using UMAP with cosine distance and `random_state=42`. Clustering is performed in this reduced feature space.

E.4 HDBSCAN Clustering

After constructing either the full NCD distance matrix or the anchor-based feature representation, we apply HDBSCAN to discover candidate latent tasks. HDBSCAN is suitable for the streaming setting because it does not require specifying the

number of clusters in advance and can explicitly mark uncertain samples as noise.

In full pairwise mode, HDBSCAN receives the precomputed NCD distance matrix and uses `metric=precomputed`. In anchor mode, HDBSCAN operates on the reduced feature representation and uses Euclidean distance. Unless otherwise specified, we use the following hyperparameters:

- `min_cluster_size` $= S_{\min}$, with $S_{\min} = 50$ by default;
- `min_samples` $= 100$, which makes cluster assignment conservative and reduces premature consolidation;
- `cluster_selection_method=eom`, corresponding to the Excess-of-Mass criterion.

Samples assigned the label -1 by HDBSCAN are treated as noise. They are not used for parametric consolidation in the current round and remain in the episodic cache for future clustering rounds.

E.5 Embedding-Based Outlier Filtering

The raw clusters returned by HDBSCAN may still contain boundary samples or semantically inconsistent samples. We therefore apply an optional embedding-based outlier filtering step to each candidate cluster. This step uses dense semantic representations to complement the surface-form similarity captured by NCD.

For a candidate cluster with index set $\mathcal{I}$, we first encode each sample using Qwen3-Embedding-0.6B and obtain ℓ_2-normalized embeddings $\{\mathbf{e}_i\}_{i\in\mathcal{I}}$. The normalized cluster centroid is computed as

$$\bar{\mathbf{e}} = \frac{\sum_{i\in\mathcal{I}} \mathbf{e}_i}{\left\|\sum_{i\in\mathcal{I}} \mathbf{e}_i\right\|_2}. \tag{18}$$

For each sample, we compute its cosine similarity to the centroid:

$$s_i = \mathbf{e}_i^{\top}\bar{\mathbf{e}}. \tag{19}$$

Let $\bar{s}$ and σ_s denote the mean and standard deviation of $\{s_i\}_{i\in\mathcal{I}}$. A sample is removed as an outlier if

$$s_i < \bar{s} - k \cdot \sigma_s, \tag{20}$$

where k controls the sensitivity of the filter. We set $k = 0.5$ by default. After filtering, if the remaining cluster size falls below $S_{\min}$, the entire cluster is discarded for the current round and its samples are retained in the episodic cache.

E.6 Cluster Quality Assessment

After outlier filtering, we further evaluate the procedural cohesion of each candidate cluster using intra-cluster NCD similarity. For a cluster $\mathcal{C}$, let $\mathcal{P}$ denote a set of unordered sample pairs drawn from $\mathcal{C}$. To reduce computation, we use all pairs when $|\mathcal{C}| \leq 15$ and otherwise randomly sample up to 50 pairs. The NCD-based cluster similarity is defined as

$$\mathrm{sim}_{\mathrm{NCD}}(\mathcal{C}) = 1 - \frac{1}{|\mathcal{P}|} \sum_{(i,j)\in\mathcal{P}} \mathrm{NCD}(x_i, x_j). \tag{21}$$

A cluster is accepted only if

$$\mathrm{sim}_{\mathrm{NCD}}(\mathcal{C}) \geq \tau_{\mathrm{cluster}}, \tag{22}$$

where the default threshold is $\tau_{\mathrm{cluster}} = 0.85$. This threshold is controlled by the hyperparameter `-cluster_quality_threshold`. Clusters with $\mathrm{sim}_{\mathrm{NCD}}(\mathcal{C}) < \tau_{\mathrm{cluster}}$ are considered low-quality and are excluded from supervised fine-tuning in the current round. Their samples remain in the episodic cache, allowing them to be reconsidered when additional evidence becomes available.

E.7 Streaming Cache Management

At the end of each clustering round, UniMem updates the episodic cache according to the clustering outcome. Samples belonging to accepted clusters are removed from the cache and passed to the procedural consolidation stage, where they are used to initialize new routing tokens and train the corresponding Procedural KV Memory blocks. Samples labeled as noise, samples removed by the embedding-based outlier filter, and samples belonging to low-quality clusters are retained in the cache.

This cache update rule is intentionally conservative. It prevents UniMem from allocating parametric memory to one-off or ambiguous patterns, while still allowing sparse tasks to accumulate enough samples across multiple streaming rounds. As a result, the system consolidates recurring and coherent task structures into parametric memory, while continuing to serve uncertain or long-tail samples through episodic retrieval.

F Computational Budget and Software Environment

Computational budget. All experiments can be run on a single NVIDIA H20 GPU with 141GB memory. For the SNI streaming experiments, the

computational cost depends mainly on the number of streaming tasks. A complete run with 100 SNI tasks takes approximately 3–4 hours on one H20 GPU. The 50-task setting takes approximately 2 hours, while the 10-task setting takes approximately 20 minutes. These estimates include streaming inference, episodic buffering, task discovery, and parametric consolidation. The reported runtime may vary slightly with backbone size, sequence length, and the number of discovered clusters.

Setting	Hardware	Approx. Runtime
SNI-10	1× NVIDIA H20 141GB	20 minutes
SNI-50	1× NVIDIA H20 141GB	2 hours
SNI-100	1× NVIDIA H20 141GB	3–4 hours

Table 6: Approximate computational budget for the main SNI streaming experiments.

Software environment. Table 7 summarizes the main libraries used in our implementation. PyTorch and Transformers are used for model training, inference, tokenization, and generation. PEFT is used for the LoRA baseline. HDBSCAN and UMAP are used in the task discovery module. Sentence-Transformers is used for embedding-based retrieval and outlier filtering, and `rouge-score` is used for ROUGE-L F1 evaluation.

Library	Version
PyTorch	2.8.0+cu128
transformers	4.57.1
peft	0.17.1
hdbscan	0.8.43
umap-learn	0.5.7
scikit-learn	1.7.2
sentence-transformers	5.1.1
rouge-score	0.1.2
numpy	1.26.4

Table 7: Main software libraries used in our experiments.

G Potential Risks

Our experiments are conducted on public NLP benchmarks and do not collect, store, or process private user data. UniMem is studied as a research prototype for memory expansion in LLM agents rather than as a deployed user-facing system. Nevertheless, persistent memory mechanisms may introduce risks in real-world applications. Incorrect or biased strategies could be consolidated and repeatedly reused if the memory update process is not properly monitored. In addition, if such a system were applied to real user interactions, sensitive information could be unintentionally retained without appropriate filtering. These risks are not present in our benchmark experiments, but they should be addressed before deployment through privacy filtering, memory auditing, and conservative consolidation policies.

H Information About Use of AI Assistants

The authors used AI assistants for language polishing, writing refinement, and clarity checking during manuscript preparation. AI assistants were not used to generate experimental results or make final scientific claims. All technical content, experimental analyses, and final writing decisions were verified and controlled by the authors. AI assistants do not meet authorship criteria and are not listed as authors.