Conditional Memory Enhanced Item Representation for Generative Recommendation

Ziwei Liu
City University of Hong Kong
Hong Kong, China

Yejing Wang
City University of Hong Kong
Hong Kong, China

Shengyu Zhou
Independent Researcher
Beijing, China

Xinhang Li
Tsinghua University
Beijing, China

Xiangyu Zhao✉
City University of Hong Kong
Hong Kong, China

Abstract

Generative recommendation (GR) has emerged as a promising paradigm that predicts target items by autoregressively generating their semantic identifiers (SID). Most GR methods follow a quantization-representation-generation pipeline, first assigning each item a SID, then constructing input representations from SID-token embeddings, and finally predicting the target SID through autoregressive generation. Existing item-level representation constructions mainly take two forms: directly merging SID-token embeddings into a compact vector, or enriching item-level representations with external inputs through additional networks. However, these item-level constructors still expose two practical challenges: direct merging may amplify the information loss caused by quantization and ID collision while obscuring SID code relations, whereas external-input-based methods can strengthen item semantics but cannot reliably preserve the SID-structured evidence required for token-level generation. These limitations make representation construction an underexplored bottleneck, leading to two severe problems, *i.e.,* the Identity-Structure Preservation Conflict and Input-Output Granularity Mismatch. To this end, we propose ComeIR, a **Co**nditional **me**mory Enhanced **I**tem **R**epresentation framework that reconstructs SID-token embeddings into item-aware inputs and restores the token granularity during SID decoding. Specifically, MM-guided token scoring adaptively estimates the contribution of each code within the SID, dual-level Engram memory captures intra-item code composition and inter-item transition patterns, and a memory-restoring prediction head reuses the memories during SID decoding. Extensive experiments demonstrate the effectiveness and flexibility of ComeIR, and further reveal scalable gains from enlarging conditional memory.

Keywords

Generative Recommendation; Conditional Memory; Large Language Model

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Ziwei Liu, Yejing Wang, Shengyu Zhou, Xinhang Li, and Xiangyu Zhao✉. 2026. Conditional Memory Enhanced Item Representation for Generative Recommendation. In . ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

With recent advances in natural language processing, large language models (LLMs) have demonstrated powerful capabilities in both semantic understanding and sequence modeling [3, 46], which has motivated generative recommendation (GR) [1, 20–22]. A general GR paradigm first assigns each item a Semantic ID (SID), *i.e.,* a tuple of discrete codes, and predicts the next item's SID from historical interactions via autoregressive generation [32]. As shown in Figure 1, this pipeline involves three stages, *i.e.,* quantization, representation, and generation. Existing research mainly focus on optimizing the first and last stages, such as learning SID with better quality [13–15, 23, 37] or designing better generators [4, 7, 24, 29, 43, 49], leaving the bridge between them, *i.e.,* the representation stage, underexplored. Yet this bridge is crucial: since the LLM only observes representations rather than SIDs, this construction determines how much quantized semantics can be preserved for generation.

The most common way to construct the representation is to map each SID's code to its corresponding token embedding and flatten all such embeddings into a token embedding sequence [20, 21, 32]. Specifically, for a user-item interaction with N items and an L-digit SID for each item, this flattening yields a GR input sequence of length $L \times N$, thereby introducing two severe limitations. First, flattening significantly increases the input length, sacrificing efficiency while increasing the risk of attention sink [12, 41]. Second, flattening obscures the item-wise structure, *i.e.,* item boundaries and intra-item code organization, carried by SID, forcing the LLM to recover them from weak positional cues rather than from an item-aware representation.

To overcome the aforementioned limitations, an intuitive solution is to construct the item-level representations. Existing works generally follow two lines. One directly merges the SID-token embeddings of each item into a compact vector, typically through a linear projection, reducing the input length from $L \times N$ to N [40, 49, 51]. The other enriches the item-level representation with external inputs, such as user behavioral signals, and uses an additional network, *e.g.,* a context compressor, to transform these heterogeneous signals into compact representations for GR [45, 49].

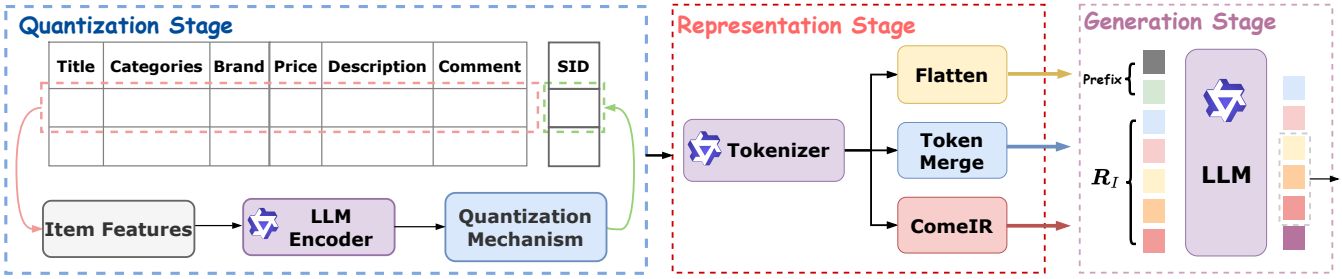


Figure 1: Overview of GR pipeline. Quantization transforms items from features to SID. Representation organizes SID as input to the LLM. Generation uses the LLM to generate the next item's SID.

Although these strategies appear effective, two challenges still undermine their practical utility. **i) Identity-Structure Preservation Conflict.** A useful item-level representation should preserve item-specific identity while retaining the structured evidence carried by SIDs. However, these two preservation goals are difficult to satisfy simultaneously under existing item-level constructors. Directly merging SID tokens keeps the input compact, but it may amplify the information loss caused by quantization and ID collision [9, 15, 16, 19], while also obscuring the code relations carried by SIDs [15, 33]. External-input-based methods can inject multimodal or behavioral contexts to strengthen item semantics, but such signals are not organized according to the structure of SID and therefore cannot reliably preserve the discrete evidence required for SID-based generation. **ii) Input-Output Granularity Mismatch.** Existing item-level constructors feed compact item vectors into the LLM, whereas the generative objective still requires token-level SID prediction for fine-grained item retrieval [51]. This forces the input and output sides to operate at different granularities, leaving the decoder to recover SID evidence from representations that have already been compressed or bypassed.

These observations call for a representation construction that is item-aware, structure-preserving, and compatible with token-level SID generation. To this end, we propose a **Co**nditional **me**mory Enhanced **I**tem **R**epresentation framework, namely **ComeIR**, to reconstruct the input of general GR models and restore the token-level granularity during SID decoding. Instead of treating SID tokens as either a flattened sequence, a simply compressed item vector, or external contexts alone, ComeIR models representation construction as a memory-conditioned transformation from SID-token embeddings to item-aware representations. Specifically, MM-guided Token Scoring uses the original multimodal item embedding to estimate the contribution of each SID code, strengthening item identity during compression. To preserve the SID-structure, we develop a dual-level Engram memory that models intra-item code composition and inter-item transition patterns via two separate sparse memories. A Memory-conditioned Token Merge then integrates the scored SID-token embeddings with the retrieved dual-level memories, injecting organized SID evidence into compact item-level representations. Finally, a Memory-restoring Prediction Head reuses the same memories during SID decoding, bridging the input-output granularity mismatch between item-level inputs and token-level generation. The contributions of this paper can be summarized as follows:

- We identify representation construction as an underexplored bottleneck in current GR, showing that existing item-level construction strategies face two practical challenges, *i.e.,* Identity-Structure Preservation Conflict and Input-Output Granularity Mismatch.
- We propose **ComeIR**, a plug-and-play conditional-memory-enhanced representation framework for generative recommendation that constructs item-aware representations while preserving the SID structures required for decoding.
- Extensive experiments on three public datasets validate the effectiveness and generality of ComeIR, while scaling analysis reveals clear log-linear scaling laws, highlighting scalable gains from enlarging the proposed conditional memory.

2 Problem Definition

The goal of GR is to directly generate the next item that a user is likely to interact with based on historical interactions. Let $\mathcal{U}$ and $\mathcal{I}$ denote the user set and item set, respectively. For each user $u \in \mathcal{U}$, the historical interactions are arranged in chronological order as follows:

$$\mathcal{S}_u = (v_1, \ldots, v_n, \ldots, v_N), \quad v_n \in \mathcal{I}, \quad n = 1, \ldots, N \tag{1}$$

where N is the sequence length. For simplicity, we omit the user index u in the following. In the quantization stage, each item is represented by a fixed-length SID rather than a single atomic ID, and the SID of item v_n is $\boldsymbol{c}_n = (c_n^1, c_n^2, \ldots, c_n^L)$, where L is the length of the SID. During the representation stage, each SID code is converted into a token embedding by combining its frozen codebook embedding with a learnable token embedding. For the ℓ-th code of item v_n,

$$\boldsymbol{e}_{c_n^\ell} = \boldsymbol{W}_E \left[\boldsymbol{e}^B_{c_n^\ell}; \boldsymbol{e}^T_{c_n^\ell} \right] \tag{2}$$

where $\boldsymbol{e}^B_{c_n^\ell}$ is the frozen codebook embedding, $\boldsymbol{e}^T_{c_n^\ell}$ is a learnable token embedding, and $\boldsymbol{W}_E$ projects their concatenation into the LLM hidden space. The SID-level representation of item v_n is $\boldsymbol{R}_n^S = \left[\boldsymbol{e}_{c_n^1}, \ldots, \boldsymbol{e}_{c_n^L} \right]$. We can then write the representation construction as $\boldsymbol{r}_n^I = f\left(\boldsymbol{R}_n^S, \boldsymbol{b}_n\right)$, and formulate the final GR input as $\boldsymbol{R}^I = \left[\boldsymbol{r}_1^I, \ldots, \boldsymbol{r}_N^I\right]$. Here, $\boldsymbol{b}_n$ denotes optional external inputs that provide additional information beyond the SID tokens. When no external inputs are introduced, $\boldsymbol{b}_n$ is omitted: for flattening, $f(\cdot)$ reduces to a simple concatenation, while for token merging, $f(\cdot)$ denotes a linear layer. For our ComeIR , we also omit $\boldsymbol{b}_n$ and leverage the existing multi-modal embedding and conditional memories to enhance our

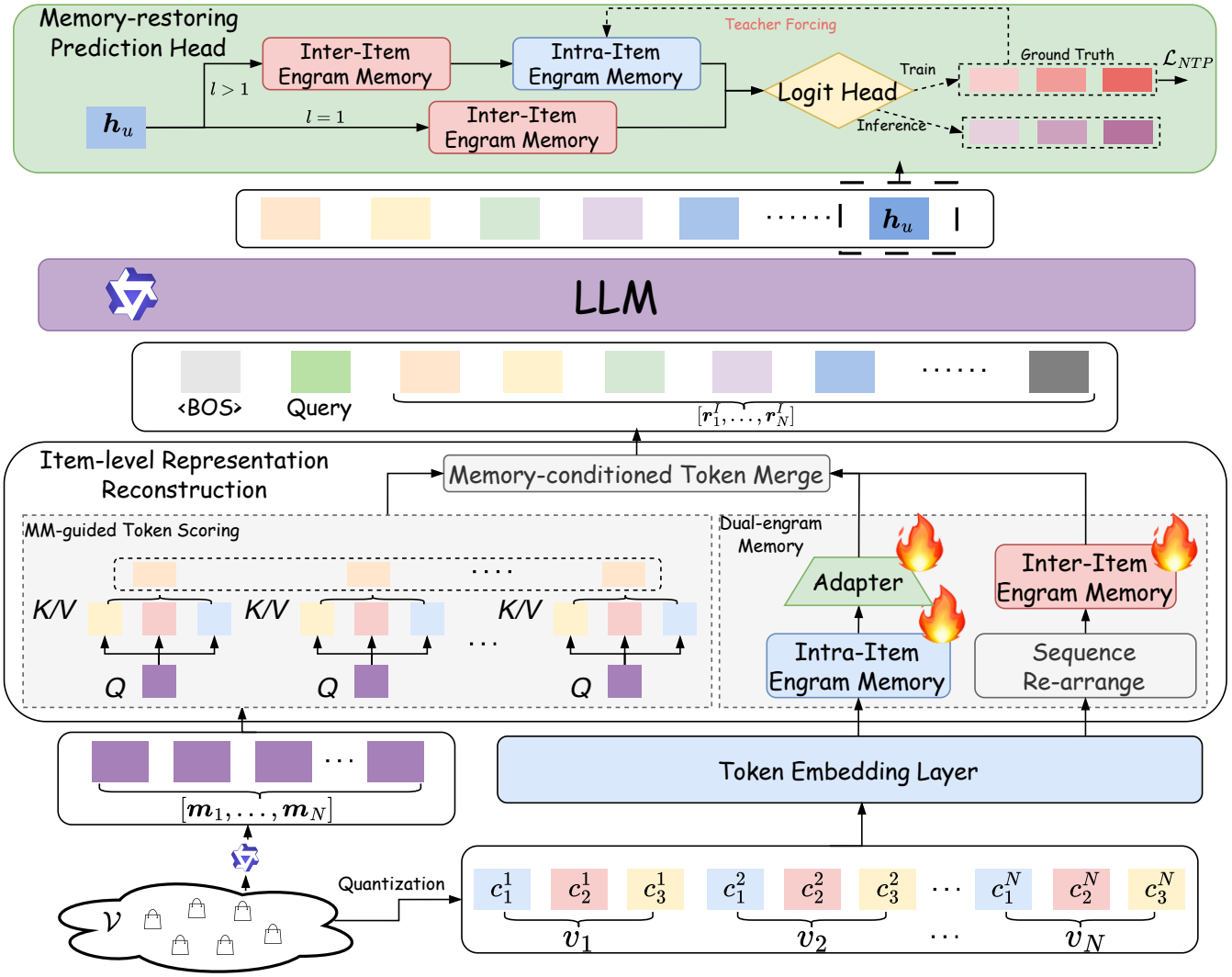


Figure 2: The overall framework of the proposed ComeIR. The code layer L is set to 3 for illustration.

representation. Finally, in the generation stage, a generative model Θ autoregressively predicts the SID of the next item based on the constructed item-level representations, which can be formulated as:

$$P\left(\boldsymbol{c}_{N+1}|\boldsymbol{R}^I;\Theta\right) = \prod_{\ell=1}^{L} P\left(c_{N+1}^{\ell}|\boldsymbol{R}^I, c_{N+1}^{<\ell};\Theta\right) \tag{3}$$

3 Method

3.1 Framework Overview

The overview of our proposed framework is illustrated in Figure 2. Given the SID-level representations $\boldsymbol{R}_n^S$ defined in Section 2, our target is to model a fine-grained $f(\cdot)$ that maps each SID into an item-level representation while preserving the structured SID evidence needed for autoregressive decoding. Accordingly, instead of directly feeding all $L \times N$ SID tokens into the LLM, ComeIR reconstructs them into N memory-conditioned item representations.

Specifically, our representation construction contains three coupled components. First, MM-guided Token Scoring reuses the cached multimodal item embeddings $\boldsymbol{M} = [\boldsymbol{m}_1, \ldots, \boldsymbol{m}_N]$, from which the SIDs are produced, to estimate the contribution of each code-layer embedding within an SID. This cached embedding acts as an identity query tied to SID construction, rather than as an external extra information. Meanwhile, Dual-level Engram Memory constructs two sparse memories over SID code patterns to retrieve intra-item code-composition evidence and inter-item transition evidence; the scored token embeddings and the two memory vectors are then aggregated via Memory-conditioned Token Merge to construct the final item-level representation sequence $\boldsymbol{R}^I$. Finally, in the generation stage, the constructed item-level representation sequence is fed into the LLM to produce $\boldsymbol{h}_u$. For layer-wise SID prediction, the Memory-restoring Prediction Head combines $\boldsymbol{h}_u$ with the intra-item and inter-item Engram contexts to reuse the token-level SID relations during decoding.

3.2 MM-guided Token Scoring

As discussed in the Identity-Structure Preservation Conflict, item-level compression should retain item-specific identity without discarding SID-structured evidence. We first address the identity side through **MM-guided Token Scoring**. The guidance comes from the same multimodal embedding used by the quantizer to produce the item's SID, so it acts as an identity query for historical items rather than an extra information source. Specifically, for item v_n, let $\boldsymbol{m}_n \in \mathbb{R}^{d_{mm}}$ denote this cached multimodal embedding. We first project it into the same dimension as the SID-token embeddings by $\boldsymbol{q}_n = \boldsymbol{W}_{\text{mm}}\boldsymbol{m}_n$, where $\boldsymbol{W}_{\text{mm}} \in \mathbb{R}^{d\times d_{mm}}$. Then, we calculate the contribution of each code in $\boldsymbol{R}_n^S$ through cross attention:

$$\alpha_n^\ell = \frac{\exp\left(\boldsymbol{q}_n^\top \boldsymbol{e}_{c_n^\ell}/\sqrt{d}\right)}{\sum_{r=1}^{L}\exp\left(\boldsymbol{q}_n^\top \boldsymbol{e}_{c_n^r}/\sqrt{d}\right)}, \quad \boldsymbol{s}_n^0 = \sum_{\ell=1}^{L}\alpha_n^\ell \boldsymbol{e}_{c_n^\ell} \tag{4}$$

Here, $\boldsymbol{s}_n^0$ is the MM-guided item context used to query memory and initialize item summary before level-wise memory injection. In this way, the representation constructor emphasizes identity-relevant codes rather than treating all codes in each item's SID equally.

3.3 Dual-level Engram Memory

To preserve the SID-structure side, we model two structures carried by SIDs. Within one item, an SID is an ordered code sequence $(c_n^1, \ldots, c_n^L)$, where later codes refine the prefix formed by previous codes. Across a user history, the level-ℓ prefixes form a discrete sequence $(\boldsymbol{c}_1^{\le\ell}, \ldots, \boldsymbol{c}_N^{\le\ell})$, whose local suffixes describe recurring preference transitions. These two forms are naturally suited to Engram-style memory: both are discrete patterns that can be repeatedly addressed, stored, and reused.

Inspired by the recent success of Engram in decoupling static memory from the LLM transformer block [6], we introduce **Dual-level Engram Memory**. Different from the original Engram module, which is inserted into Transformer blocks, our memory is placed at the representation interface: it retrieves SID-pattern evidence before item-level inputs are built and restores the same type of evidence during token-level prediction. In this section, we will first define a general Engram read function, then instantiate it for intra-item code composition and inter-item transition modeling.

General Engram. The general engram receives two inputs, *i.e.,* a discrete code pattern $\boldsymbol{p}$ and a contextual query $\boldsymbol{q}$, which respectively represent what to look up and when the look-up evidence should be trusted. To be more specific, we divide the whole process into two steps:

Step 1: Hashed N-gram lookup. To perform the N-gram hash lookup, we first extract suffix N-gram keys of multiple orders from $\boldsymbol{p} = (p_1, \ldots, p_T)$. An order-$o$ key consists of the last o elements $(p_{T-o+1}, \ldots, p_T)$ from $\boldsymbol{p}$, where each element is defined by the memory-specific pattern introduced below. We denote the set of orders at level ℓ as O_ℓ. Each order-o key is hashed into K independent sparse embedding tables via deterministic multi-head hashing [34], and the K vectors are concatenated into one address $\boldsymbol{a}_{\ell,o}(\boldsymbol{p}) \in \mathbb{R}^{d_m}$. Since the same discrete pattern is deterministically mapped to the same table rows across training examples, the corresponding retrieved address can serve as a reusable memory slot for that recurring SID pattern.

Step 2: Context-aware gating. Considering the N-gram code combinations are static, which inherently lack contextual adaptability and may suffer from noise due to hash collisions or polysemy [6, 47], a context-aware gate is needed to filter the noise according to the corresponding context. Specifically, we devise a scalar gate $\lambda_{\ell,o} \in (0, 1)$ to score the compatibility between $\boldsymbol{q}$ and each address (details in Appendix A.1). Finally, the Engram memory is calculated by aggregating all gated addresses as :

$$\mathcal{R}_\ell(\boldsymbol{q}, \boldsymbol{p}) = \text{LN}\left(\sum_{o\in O_\ell}\lambda_{\ell,o}\boldsymbol{W}_{V,\ell,o}\boldsymbol{a}_{\ell,o}(\boldsymbol{p})\right) \tag{5}$$

Here $\boldsymbol{W}_{V,\ell,o}$ projects each address into memory evidence, and $\text{LN}(\cdot)$ normalizes the sum. Table capacities and hashing details are presented in Appendix B.2.2.

Intra-item Engram. The intra-item Engram memorizes how codes are composed within a single SID. For example, an SID such as (c_n^1, c_n^2, c_n^3), the pattern (c_n^1, c_n^2) describes how the second code specializes the first-level code. This conditional-prefix form matches the suffix lookup above because the key should emphasize the latest refinement under its preceding SID context. Formally, for item v_n, define $\boldsymbol{c}_n^{<\ell} = (c_n^1, \ldots, c_n^{\ell-1})$ and $\boldsymbol{c}_n^{\le\ell} = (c_n^1, \ldots, c_n^\ell)$. Since the first code has no preceding prefix, intra-item patterns start from level $\ell = 2$. At each level, we store the conditional pattern $\boldsymbol{p}_{n,\ell}^S = (\boldsymbol{c}_n^{<\ell}, c_n^\ell)$, which records how the ℓ-th code refines its preceding prefix. Using $\boldsymbol{s}_n^0$ as the query, we read the conditional evidence at the same level ℓ and compose accumulated reads into a single intra-item memory as follows:

$$\boldsymbol{z}_{n,\ell}^S = \mathcal{R}_\ell\left(\boldsymbol{s}_n^0, \boldsymbol{p}_{n,\ell}^S\right), \quad \boldsymbol{\eta}_{n,\ell}^S = \boldsymbol{W}_{C,\ell}^S\left[\boldsymbol{z}_{n,2}^S; \ldots; \boldsymbol{z}_{n,\ell}^S\right] + \boldsymbol{b}_{C,\ell}^S, \quad \ell = 2, \ldots, L \tag{6}$$

The adapter $(\boldsymbol{W}_{C,\ell}^S, \boldsymbol{b}_{C,\ell}^S)$ compresses the collected reads into compact SID-composition evidence $\boldsymbol{\eta}_{n,\ell}^S$ for Token Merge. Further details are in Appendix A.2.

Inter-item Engram. The inter-item Engram captures transition patterns across user-item interactions at multiple granularities. Specifically, for each code level ℓ, we arrange all historical ℓ-level SID prefixes into a sequence $C^{\le\ell} = (\boldsymbol{c}_1^{\le\ell}, \ldots, \boldsymbol{c}_N^{\le\ell})$. Shallow prefixes ($\ell$=1) track broad preference changes, *e.g.,* different categories; deeper prefixes capture finer transitions, *e.g.,* preference towards specific items. The sequence rearrangement only changes the unit being read by the Engram: each element in $C^{\le\ell}$ is the prefix $\boldsymbol{c}_a^{\le\ell}$ of an interacted item, and the chronological order is still preserved. The transition pattern for item v_n can be then defined as $\boldsymbol{p}_{n,\ell}^T = \text{Suffix}_{\tau_\ell}\left(C^{\le\ell}, n\right)$, where the suffix keeps the most recent level-ℓ prefix transitions before and including position n. Detailed constructions are presented in Equation (20).

Since this Engram captures inter-item transition, a single item's context is insufficient. We therefore construct a transition-aware query $\boldsymbol{q}_{n,\ell}^T$ from a local window of MM-guided item contexts (detailed in Appendix A.3) and retrieve the inter-item memory as: $\boldsymbol{\eta}_{n,\ell}^T = \mathcal{R}_\ell\left(\boldsymbol{q}_{n,\ell}^T, \boldsymbol{p}_{n,\ell}^T\right)$.

3.4 Memory-conditioned Token Merge

For a specific item v_n, after obtaining the MM-guided item context $\boldsymbol{s}_n^0$ and dual-level memories, *i.e.,* $\boldsymbol{\eta}_{n,\ell}^S$ and $\boldsymbol{\eta}_{n,\ell}^T$ at each code level ℓ, we need to transform multiple signals into one item-level input.

However, a naive merge collapses all code embeddings at once and cannot recognize item identity or the SID structure. Unlike external-input-based constructors that enrich item vectors without explicitly organizing SID evidence, we propose the **Memory-conditioned Token Merge** that performs level-wise gated updates to ensure the retrieved memories are beneficial according to the current item context. Specifically, at each level $\ell = 1, \ldots, L$, the dual-level memories are stacked into $\boldsymbol{u}_{n,\ell} = [\bar{\boldsymbol{\eta}}^S_{n,\ell}; \boldsymbol{\eta}^T_{n,\ell}]$ (with $\bar{\boldsymbol{\eta}}^S_{n,1}$=$\mathbf{0}$ since intra-item evidence only exists from level 2 onward). A scalar gate $\omega_{n,\ell} \in (0, 1)$ then measures whether the current summary $\boldsymbol{s}_n^{\ell-1}$ is compatible with the memory $\boldsymbol{u}_{n,\ell}$, and the summary is updated via a gated residual:

$$\omega_{n,\ell} = \sigma\left(\left(\boldsymbol{W}^{\mathrm{M}}_{Q}\boldsymbol{s}_n^{\ell-1}\right)^{\top}\left(\boldsymbol{W}^{\mathrm{M}}_{K,\ell}\boldsymbol{u}_{n,\ell}\right) + b^{\mathrm{M}}_{\ell}\right), \quad \boldsymbol{s}_n^{\ell} = \boldsymbol{s}_n^{\ell-1} + \omega_{n,\ell}\boldsymbol{W}^{\mathrm{M}}_{V,\ell}\boldsymbol{u}_{n,\ell} \tag{7}$$

After level L, we set $\boldsymbol{r}^I_n = \mathrm{LN}(\boldsymbol{s}^L_n)$ and collect $\boldsymbol{R}^I = [\boldsymbol{r}^I_1, \ldots, \boldsymbol{r}^I_N]$ as the final item-level GR input.

3.5 Memory-restoring Prediction Head

While item-level input improves efficiency, the target item still needs to be generated as a token-level SID during decoding for fine-grained item retrieval [51]. Consequently, we design the **Memory-restoring Prediction Head**, which restores intra-item and inter-item memory during SID decoding.

Specifically, when predicting the ℓ-th code, we denote the generated prefix as $\boldsymbol{c}^{<\ell}_{N+1}$ and the candidate code as x. The head restores memory evidence in a layer-dependent manner:

First code (ℓ=1). No intra-item prefix exists yet, so we set the intra-item evidence to $\bar{\boldsymbol{\eta}}^S_{N+1,1}(x) = \mathbf{0}$ and rely solely on inter-item transition evidence. The inter-item memory is retrieved by appending the candidate (x) to the historical prefix sequence and reading from the Engram:

$$\boldsymbol{\mu}_1(x) = \boldsymbol{W}_{T,1}\,\mathcal{R}_1\left(\boldsymbol{q}^{\mathrm{D}}_{u,1},\ \boldsymbol{p}^T_{N+1,1}(x)\right) \tag{8}$$

where $\boldsymbol{q}^{\mathrm{D}}_{u,\ell}$ is a transition query constructed from $\boldsymbol{h}_u$ and the recent item contexts.

Subsequent codes ($\ell \geq 2$). Once a partial prefix $\boldsymbol{c}^{<\ell}_{N+1}$ is available, the head first reads intra-item evidence from the conditional pattern $(\boldsymbol{c}^{<\ell}_{N+1}, x)$, which captures how code x refines the generated prefix within the target SID. This evidence is then supplemented by inter-item transition evidence:

$$\boldsymbol{\mu}_\ell(x) = \boldsymbol{W}_{S,\ell}\,\mathcal{R}_\ell\left(\boldsymbol{h}_u,\ (\boldsymbol{c}^{<\ell}_{N+1}, x)\right) + \boldsymbol{W}_{T,\ell}\,\mathcal{R}_\ell\left(\boldsymbol{q}^{\mathrm{D}}_{u,\ell},\ \boldsymbol{p}^T_{N+1,\ell}(x)\right) \tag{9}$$

In both cases, $\boldsymbol{\mu}_\ell(x)$ is fused with the user state $\boldsymbol{h}_u$ and the candidate embedding $\boldsymbol{e}_x$ to produce a logit for x, which can be formulated as:

$$\boldsymbol{d}_\ell(x) = \mathrm{LN}\left(\boldsymbol{W}_{H,\ell}\boldsymbol{h}_u + \boldsymbol{W}_{C,\ell}\boldsymbol{e}_x + \boldsymbol{\mu}_\ell(x)\right), \quad \psi_\ell(x) = \boldsymbol{w}_\ell^{\top}\boldsymbol{d}_\ell(x) \tag{10}$$

The layer-wise probability is normalized over the set of catalog-valid codes $\mathcal{V}_\ell(\boldsymbol{c}^{<\ell}_{N+1})$, determined by a prefix tree built from all SIDs in the catalog:

$$P\left(c^\ell_{N+1} = x \middle| \boldsymbol{R}^I, \boldsymbol{c}^{<\ell}_{N+1}; \Theta\right) = \frac{\exp(\psi_\ell(x))}{\sum_{y \in \mathcal{V}_\ell(\boldsymbol{c}^{<\ell}_{N+1})}\exp(\psi_\ell(y))} \tag{11}$$

More details about the implementations on different architectures can be found in Appendix A.4.

3.6 Training and Inference

Training. Given the ground-truth next-item SID $\boldsymbol{c}_{N+1}$, we train ComeIR with teacher forcing and token-level cross-entropy over SID layers. At layer ℓ, the ground-truth prefix $\boldsymbol{c}^{<\ell}_{N+1}$ is provided to the prediction head, so the model learns to score the next code under valid historical and prefix-conditioned memory evidence. The training loss can then be defined as:

$$\mathcal{L}_{\mathrm{rec}} = -\sum_{u \in \mathcal{U}}\sum_{\ell=1}^{L}\log P\left(c^\ell_{N+1} \middle| \boldsymbol{R}^I, \boldsymbol{c}^{<\ell}_{N+1}; \Theta\right) \tag{12}$$

Inference. During inference, ComeIR first constructs $\boldsymbol{R}^I$ and obtains $\boldsymbol{h}_u$ from the LLM. It then performs beam search over SID layers, while each candidate is scored by restoring the same intra-item and inter-item memories used in training. Detailed settings for different architectures, *i.e.,* normal GR and NEZHA, can be found in Appendix B.3.

4 Experiment

4.1 Experimental Settings

Dataset. There are three public datasets applied for evaluation, *i.e.,* Yelp, Amazon Industrial, and Amazon Instrument. More details can be found in the Appendix B.1.

Baselines & Backbones. We construct our experiments leveraging different quantization mechanisms, *i.e.,* RQ-VAE [19] and RQ-Kmeans [16], different LLM backbones, *i.e.,* Qwen3-0.6B and LLaMA3-1B, and different architectures, *i.e.,* normal GR and NEZHA [39]. The '*+ TM*' denotes token merging, following the settings of previous work [51], which naively concatenates all code embeddings for each item and leverages a linear layer to project back to the LLM's hidden size.

Evaluation Metrics. We adopt the commonly used metrics, *i.e., hit rate* (H@K) and *Normalized Discounted Cumulative Gain* (N@K), truncated at K, where $K \in 5, 10$. For efficiency, we also report the generation latency (LT) in milliseconds (ms) per batch, following the previous work [39].

Implementation Details. For general settings, *e.g.,* the intra-item and inter-item Engram tables, we set a base of 128 and 16, respectively. The two scaling parameters (intra & inter) for enlarging the table scale are 1.0 and 2.0 by default. Other details can be found in the Appendix B.2.

4.2 Overall Performance

To validate the effectiveness and flexibility of the proposed ComeIR, we compare its performance across various quantization mechanisms, LLM backbones, and architectures, both with the general setting (flattening the sequence) and with token merging. As shown in Table 1, ComeIR demonstrates superior recommendation performance and remarkable robustness. By integrating ComeIR into current architectures (**+ ComeIR**), significant gains are consistently achieved across all commonly adopted quantization mechanisms, *i.e.,* RQ-VAE and RQ-KMeans, and both backbones. Moreover, the comparison with *+ TM* further shows that simply compression is insufficient; memory-conditioned construction is necessary to preserve fine-grained code evidence during item-level compression. Specifically, on the Yelp dataset with RQ-VAE, ComeIR yields its

Table 1: Overall performance of ComeIR under different settings. bold values are the best, and "*" marks significant gains (one-side t-test with p<0.05) over the matched architecture.

(a) Performance with Qwen3-0.6B as the LLM backbone.

Quantization	Method	Yelp				Industrial				Instrument			
		H@5	H@10	N@5	N@10	H@5	H@10	N@5	N@10	H@5	H@10	N@5	N@10
RQ-VAE	**Normal GR**	0.0296	0.0474	0.0181	0.0243	0.0979	0.1275	0.0745	0.0844	0.0804	0.0982	0.0688	0.0735
	+ *TM*	0.0292	0.0470	0.0177	0.0239	0.0976	0.1271	0.0742	0.0841	0.0788	0.0966	0.0672	0.0719
	+ *ComeIR*	**0.0305***	**0.0524***	**0.0198***	**0.0266***	**0.1031***	**0.1321***	**0.0792***	**0.0881***	**0.0834***	**0.1021***	**0.0709***	**0.0772***
	NEZHA	0.0294	0.0473	0.0176	0.0241	0.0981	0.1278	0.0749	0.0847	0.0801	0.0977	0.0684	0.0730
	+ *TM*	0.0287	0.0466	0.0169	0.0234	0.0977	0.1274	0.0746	0.0844	0.0784	0.0960	0.0667	0.0713
	+ *ComeIR*	**0.0302***	**0.0520***	**0.0194***	**0.0263***	**0.1035***	**0.1324***	**0.0796***	**0.0883***	**0.0831***	**0.1015***	**0.0705***	**0.0767***
RQ-KMeans	**Normal GR**	0.0300	0.0481	0.0192	0.0251	0.1005	0.1299	0.0771	0.0870	0.0817	0.0999	0.0705	0.0754
	+ *TM*	0.0295	0.0476	0.0187	0.0246	0.1001	0.1295	0.0767	0.0867	0.0801	0.0983	0.0689	0.0738
	+ *ComeIR*	**0.0309***	**0.0530***	**0.0210***	**0.0274***	**0.1059***	**0.1345***	**0.0819***	**0.0908***	**0.0848***	**0.1039***	**0.0727***	**0.0792***
	NEZHA	0.0288	0.0476	0.0187	0.0248	0.1008	0.1299	0.0775	0.0872	0.0814	0.0996	0.0702	0.0753
	+ *TM*	0.0282	0.0470	0.0181	0.0242	0.1004	0.1295	0.0771	0.0869	0.0797	0.0979	0.0685	0.0736
	+ *ComeIR*	**0.0297***	**0.0524***	**0.0204***	**0.0271***	**0.1062***	**0.1345***	**0.0823***	**0.0909***	**0.0845***	**0.1035***	**0.0723***	**0.0791***

(b) Performance with LLaMA3-1B as the LLM backbone.

Quantization	Method	Yelp				Industrial				Instrument			
		H@5	H@10	N@5	N@10	H@5	H@10	N@5	N@10	H@5	H@10	N@5	N@10
RQ-VAE	**Normal GR**	0.0216	0.0373	0.0131	0.0182	0.0937	0.1223	0.0701	0.0794	0.0748	0.0932	0.0646	0.0696
	+ *TM*	0.0205	0.0362	0.0120	0.0171	0.0932	0.1218	0.0696	0.0788	0.0731	0.0915	0.0629	0.0679
	+ *ComeIR*	**0.0219***	**0.0416***	**0.0143***	**0.0199***	**0.0987***	**0.1267***	**0.0746***	**0.0829***	**0.0777***	**0.0970***	**0.0666***	**0.0732***
	NEZHA	0.0212	0.0372	0.0126	0.0177	0.0941	0.1223	0.0703	0.0798	0.0747	0.0929	0.0644	0.0693
	+ *TM*	0.0200	0.0360	0.0114	0.0165	0.0935	0.1217	0.0697	0.0792	0.0729	0.0911	0.0626	0.0675
	+ *ComeIR*	**0.0215***	**0.0414***	**0.0138***	**0.0194***	**0.0992***	**0.1267***	**0.0748***	**0.0833***	**0.0776***	**0.0966***	**0.0664***	**0.0728***
RQ-KMeans	**Normal GR**	0.0225	0.0393	0.0150	0.0209	0.0961	0.1243	0.0722	0.0821	0.0757	0.0949	0.0662	0.0716
	+ *TM*	0.0216	0.0384	0.0141	0.0200	0.0958	0.1240	0.0719	0.0818	0.0739	0.0931	0.0644	0.0698
	+ *ComeIR*	**0.0230***	**0.0438***	**0.0164***	**0.0228***	**0.1013***	**0.1289***	**0.0769***	**0.0858***	**0.0786***	**0.0987***	**0.0681***	**0.0752***
	NEZHA	0.0218	0.0387	0.0149	0.0202	0.0962	0.1248	0.0727	0.0826	0.0754	0.0944	0.0660	0.0711
	+ *TM*	0.0208	0.0377	0.0139	0.0192	0.0959	0.1245	0.0724	0.0823	0.0736	0.0926	0.0642	0.0693
	+ *ComeIR*	**0.0223***	**0.0431***	**0.0163***	**0.0221***	**0.1015***	**0.1294***	**0.0774***	**0.0863***	**0.0784***	**0.0982***	**0.0680***	**0.0747***

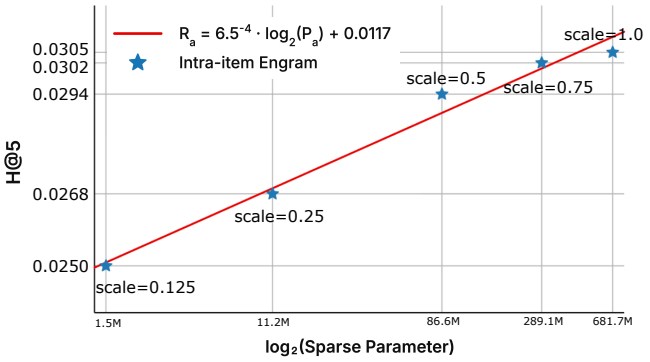
largest average improvements over different architectures: **8.06%** for Qwen3-0.6B and **7.91%** for LLaMA3-1B. The gains are more moderate in Industrial and Instrument, as expected under the fixed-scale setting in Section 4.1: larger datasets reduce the effective memory capacity, thereby increasing hash collisions and limiting the marginal gain. Overall, the consistent performance gains of ComeIR are remarkable, making it a plug-and-play framework that can be seamlessly integrated into various GR pipelines.

4.3 Ablation Study

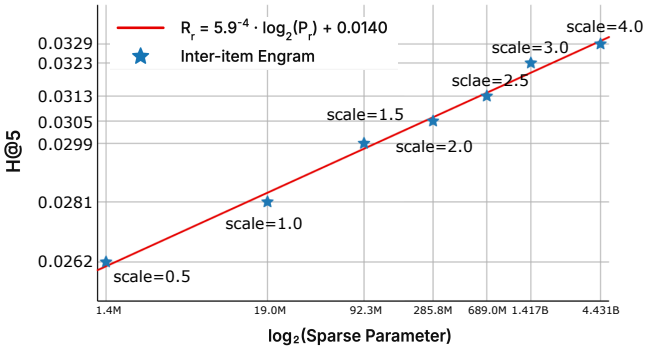
The results of the ablation study are shown in Table 2. Firstly, removing MM-guided Token Scoring leads to consistent drops, with H@5 and N@5 decreasing by 3.61% and 4.04%, respectively. This indicates that SID tokens contribute unequally to item identity. The decline of *w/o* Mem. Merge further underscores the need to inject structured SID information during representation construction. We also observe that removing memory evidence from the decoding

Table 2: Ablation results on Yelp dataset. *w/o* MM-Scoring replaces original module with mean pooling, and *w/o* Mem. Merge replaces the original module with a linear layer. Other variants remove intra-item or inter-item memory from the encoding (E) or decoding (D).

Variant	H@5	H@10	N@5	N@10
ComeIR	**0.0305**	**0.0524**	**0.0198**	**0.0266**
w/o MM-scoring	0.0294	0.0517	0.0190	0.0254
w/o **D**-intra	0.0294	0.0501	0.0193	0.0257
w/o **D**-inter	0.0291	0.0505	0.0188	0.0251
w/o Mem. Merge	0.0298	0.0512	0.0193	0.0259
w/o **E**-intra	0.0297	0.0509	0.0194	0.0244
w/o **E**-inter	0.0293	0.0516	0.0185	0.0249



(a) Intra-item Engram



(b) Inter-item Engram

Figure 3: Scaling analysis of dual-level Engram memory on the Yelp dataset.

side harms the performance. In particular, *w/o* **D**-intra and *w/o* **D**-inter underperform the full model by 3.38% and 5.64% on N@10, respectively. Such changes suggest that item-level inputs alone cannot fully recover token-level SID evidence, highlighting the need for memory restoration during layer-wise decoding. Finally, the drops of *w/o* **E**-intra and *w/o* **E**-inter validate the effect of Dual-level Engram Memory, where intra-item memory preserves code composition and inter-item memory captures historical transitions. Overall, these variants demonstrate that both representation-side memory construction and decoding-side memory restoration are indispensable to ComeIR. More results are provided in Appendix C.1.

4.4 Scaling Analysis

Following the protocols in previous work [30, 31], we also examine whether the two sparse memories, *i.e.,* intra-item and inter-item engrams, compile to a similar scaling law. The results are plotted in Figure 3. For both memories, we observe a clear power-law scaling across different scaling parameters, consistent with previous findings [6]. These results confirm the scalability of the proposed dual-level Engram memory: increasing sparse capacity reduces hash collisions and continuously improves the quality of our representation construction. More details are provided in Appendix C.2.

Table 3: Inference latency (LT) of ComeIR per batch across different datasets, LLM backbones, and architectures. We set the batch size to 32 and the beam size to 20 for all baselines for fair comparison.

Dataset	Method	Qwen3-0.6B		LLaMA3-1B	
		Normal GR LT ↓	NEZHA LT ↓	Normal GR LT ↓	NEZHA LT ↓
Yelp	Flattened	23.673	1.126	35.483	1.704
	+ ComeIR	**9.161**	**0.438**	**14.322**	**0.682**
Industrial	Flattened	21.406	1.018	32.109	1.538
	+ ComeIR	**9.081**	**0.432**	**14.176**	**0.674**
Instrument	Flattened	26.375	1.253	39.563	1.894
	+ ComeIR	**8.874**	**0.443**	**14.845**	**0.680**

4.5 Efficiency Analysis

As previously discussed, the item-level representation significantly reduces the length for GR input from $N \times L$ to N. To further estimate that, we report the inference latency of ComeIR in Table 3. The results show that, by optimizing the representation stage rather than quantization or generation, our proposed ComeIR achieves a remarkable **2.5**× average speedup through length reduction, even with the efficient NEZHA architecture. This phenomenon further validates that employing ComeIR in the current GR pipeline can not only improve effectiveness but also efficiency.

5 Related Works

Generative Recommendation. Generative recommendation (GR) reformulates item retrieval as autoregressive identifier generation conditioned on user histories. P5 [10] casts recommendation tasks into language processing, and TIGER [32] establishes the representative Semantic ID (SID) pipeline, where each item is encoded as SID, and the LLM directly generates the next item. Following the quantization-generation pipeline [7, 26, 29, 42, 44, 49], a line of research improve SID construction with content, collaborative signals, or task-aware tokenization [13, 27, 37, 38, 50], while others study SID-language alignment, long-SID generation, inductive decoding, and SID redistribution [8, 14, 40, 48]. While these works advance SID and generator design, ComeIR studies a less-explored representation bridge that reconstructs SID-token embeddings into item-aware inputs and restores them for token-level prediction.

Conditional Memory. Memory-based modeling reuses recurring patterns through explicit or implicit storage. Classical n-gram language models store local statistics [17, 18], neural n-gram and hash embeddings encode reusable evidence as compact vectors [34, 47], and modern language models exhibit memory-like behavior through key-value feed-forward layers or retrieval-augmented generation [5, 11, 35]. Conditional computation and Mixture-of-Experts improve scaling by sparse activation [2, 36], while Engram [6] introduces conditional memory as sparse lookup over n-gram patterns, with follow-up work studying its serving and indexing properties [25, 28]. Our use of conditional memory differs in both object and role: instead of storing linguistic patterns, ComeIR builds dual-level memories over two different SID patterns, capturing intra-item

code composition and inter-item code transitions to preserve SID structure during item-level compression and restore token-level granularities during generation.

6 Conclusion

In this paper, we identify representation construction as a key bottleneck in current GR pipelines, where existing item-level constructors face an identity-structure preservation conflict and an input-output granularity mismatch. Consequently, we propose a conditional memory-enhanced item representation framework (ComeIR) that uses MM-guided token scoring to strengthen item identity, dual-level Engram memories to preserve SID structure, and memory-conditioned token merging to construct compact item-level inputs. A memory-restoring prediction head further reuses these memories during SID decoding, bridging item-level inputs with token-level generation. Extensive experiments demonstrate the effectiveness, flexibility, and scalability of ComeIR.

References

[1] Yimeng Bai, Chang Liu, Yang Zhang, Dingxian Wang, Frank Yang, Andrew Rabinovich, Wenge Rong, and Fuli Feng. 2025. Bi-Level Optimization for Generative Recommendation: Bridging Tokenization and Generation. *arXiv preprint arXiv:2510.21242* (2025).

[2] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. 2013. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432* (2013).

[3] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM transactions on intelligent systems and technology* 15, 3 (2024), 1–45.

[4] Zerui Chen, Heng Chang, Tianying Liu, Chuantian Zhou, Yi Cao, Jiandong Ding, Ming Liu, and Bing Qin. 2026. Beyond the Flat Sequence: Hierarchical and Preference-Aware Generative Recommendations. In *Proceedings of the ACM Web Conference 2026*. 7999–8007.

[5] Xin Cheng, Di Luo, Xiuying Chen, Lemao Liu, Dongyan Zhao, and Rui Yan. 2023. Lift yourself up: Retrieval-augmented text generation with self-memory. *Advances in Neural Information Processing Systems* 36 (2023), 43780–43799.

[6] Xin Cheng, Wangding Zeng, Damai Dai, Qinyu Chen, Bingxuan Wang, Zhenda Xie, Kezhao Huang, Xingkai Yu, Zhewen Hao, Yukun Li, et al. 2026. Conditional memory via scalable lookup: A new axis of sparsity for large language models. *arXiv preprint arXiv:2601.07372* (2026).

[7] Jiaxin Deng, Shiyao Wang, Kuo Cai, Lejian Ren, Qigen Hu, Weifeng Ding, Qiang Luo, and Guorui Zhou. 2025. Onerec: Unifying retrieve and rank with generative recommender and iterative preference alignment. *arXiv preprint arXiv:2502.18965* (2025).

[8] Yijie Ding, Jiacheng Li, Julian McAuley, and Yupeng Hou. 2026. Inductive generative recommendation via retrieval-based speculation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 40. 14675–14683.

[9] Dengzhao Fang, Jingtong Gao, Chengcheng Zhu, Yu Li, Xiangyu Zhao, and Yi Chang. 2025. Hid-vae: Interpretable generative recommendation via hierarchical and disentangled semantic ids. *arXiv preprint arXiv:2508.04618* (2025).

[10] Shijie Geng, Shuchang Liu, Zuohui Fu, Yingqiang Ge, and Yongfeng Zhang. 2022. Recommendation as language processing (rlp): A unified pretrain, personalized prompt & predict paradigm (p5). In *Proceedings of the 16th ACM conference on recommender systems*. 299–315.

[11] Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. 2021. Transformer feed-forward layers are key-value memories. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 5484–5495.

[12] Xiangming Gu, Tianyu Pang, Chao Du, Qian Liu, Fengzhuo Zhang, Cunxiao Du, Ye Wang, and Min Lin. 2024. When attention sink emerges in language models: An empirical view. *arXiv preprint arXiv:2410.10781* (2024).

[13] Yupeng Hou, Zhankui He, Julian McAuley, and Wayne Xin Zhao. 2023. Learning vector-quantized item representation for transferable sequential recommenders. In *Proceedings of the ACM Web Conference 2023*. 1162–1171.

[14] Yupeng Hou, Jiacheng Li, Ashley Shin, Jinsung Jeon, Abhishek Santhanam, Wei Shao, Kaveh Hassani, Ning Yao, and Julian McAuley. 2025. Generating long semantic ids in parallel for recommendation. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*. 956–966.

[15] Zheng Hu, Yuxin Chen, Yongsen Pan, Xu Yuan, Yuting Yin, Daoyuan Wang, Boyang Xia, Zefei Luo, Hongyang Wang, Songhao Ni, et al. 2026. Stop Treating Collisions Equally: Qualification-Aware Semantic ID Learning for Recommendation at Industrial Scale. *arXiv preprint arXiv:2603.00632* (2026).

[16] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

[17] Slava Katz. 1987. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE transactions on acoustics, speech, and signal processing* 35, 3 (1987), 400–401.

[18] Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *1995 international conference on acoustics, speech, and signal processing*, Vol. 1. IEEE, 181–184.

[19] Doyup Lee, Chiheon Kim, Saehoon Kim, Minsu Cho, and Wook-Shin Han. 2022. Autoregressive image generation using residual quantization. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 11523–11532.

[20] Lei Li, Yongfeng Zhang, Dugang Liu, and Li Chen. 2024. Large language models for generative recommendation: A survey and visionary discussions. In *Proceedings of the 2024 joint international conference on computational linguistics, language resources and evaluation (LREC-COLING 2024)*. 10146–10159.

[21] Xiaopeng Li, Bo Chen, Junda She, Shiteng Cao, You Wang, Qinlin Jia, Haiying He, Zheli Zhou, Zhao Liu, Ji Liu, et al. 2025. A Survey of Generative Recommendation from a Tri-Decoupled Perspective: Tokenization. *Architecture, and Optimization* (2025).

[22] Yongqi Li, Xinyu Lin, Wenjie Wang, Fuli Feng, Liang Pang, Wenjie Li, Liqiang Nie, Xiangnan He, and Tat-Seng Chua. 2024. A survey of generative search and recommendation in the era of large language models. *arXiv preprint arXiv:2404.16924* (2024).

[23] Zhao Li, FengYang Qi, Chuanyu Xu, Tao Zhang, Chengfu Huo, and Peng Zhang. 2026. LSIG: Long Semantic IDs for Generative Recommendation. In *Proceedings of the ACM Web Conference 2026*. 7779–7788.

[24] Jiacheng Lin, Tian Wang, and Kun Qian. 2025. Rec-r1: Bridging generative large language models and user-centric recommendation systems via reinforcement learning. *arXiv preprint arXiv:2503.24289* (2025).

[25] Tao Lin. 2026. A Collision-Free Hot-Tier Extension for Engram-Style Conditional Memory: A Controlled Study of Training Dynamics. *arXiv preprint arXiv:2601.16531* (2026).

[26] Xinyu Lin, Chaoqun Yang, Wenjie Wang, Yongqi Li, Cunxiao Du, Fuli Feng, See-Kiong Ng, and Tat-Seng Chua. 2024. Efficient inference for large language model-based generative recommendation. *arXiv preprint arXiv:2410.05165* (2024).

[27] Enze Liu, Bowen Zheng, Cheng Ling, Lantao Hu, Han Li, and Wayne Xin Zhao. 2024. End-to-end learnable item tokenization for generative recommendation. *arXiv preprint arXiv:2409.05546* (2024).

[28] Ruiyang Ma, Teng Ma, Zhiyuan Su, Hantian Zha, Xinpeng Zhao, Xuchun Shang, Xingrui Yi, Zheng Liu, Zhu Cao, An Wu, et al. 2026. Pooling Engram Conditional Memory in Large Language Models using CXL. In *Proceedings of the Sixth European Workshop on Machine Learning and Systems*. 225–231.

[29] Kidist Amde Mekonnen, Yubao Tang, and Maarten de Rijke. 2026. A Parametric Memory Head for Continual Generative Retrieval. *arXiv preprint arXiv:2604.23388* (2026).

[30] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[31] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[32] Shashank Rajput, Nikhil Mehta, Anima Singh, Raghunandan Hulikal Keshavan, Trung Vu, Lukasz Heldt, Lichan Hong, Yi Tay, Vinh Tran, Jonah Samost, et al. 2023. Recommender systems with generative retrieval. *Advances in Neural Information Processing Systems* 36 (2023), 10299–10315.

[33] Anima Singh, Trung Vu, Nikhil Mehta, Raghunandan Keshavan, Maheswaran Sathiamoorthy, Yilin Zheng, Lichan Hong, Lukasz Heldt, Li Wei, Devansh Tandon, et al. 2024. Better generalization with semantic ids: A case study in ranking for recommendations. In *Proceedings of the 18th ACM Conference on Recommender Systems*. 1039–1044.

[34] Dan Tito Svenstrup, Jonas Hansen, and Ole Winther. 2017. Hash embeddings for efficient word representations. *Advances in neural information processing systems* 30 (2017).

[35] Boxin Wang, Wei Ping, Peng Xu, Lawrence McAfee, Zihan Liu, Mohammad Shoeybi, Yi Dong, Oleksii Kuchaiev, Bo Li, Chaowei Xiao, et al. 2023. Shall we pretrain autoregressive language models with retrieval? a comprehensive study. In *Proceedings of the 2023 conference on empirical methods in natural language processing*. 7763–7786.

[36] Lean Wang, Huazuo Gao, Chenggang Zhao, Xu Sun, and Damai Dai. 2024. Auxiliary-loss-free load balancing strategy for mixture-of-experts. *arXiv preprint arXiv:2408.15664* (2024).

[37] Wenjie Wang, Honghui Bao, Xinyu Lin, Jizhi Zhang, Yongqi Li, Fuli Feng, See-Kiong Ng, and Tat-Seng Chua. 2024. Learnable item tokenization for generative recommendation. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*. 2400–2409.

[38] Ye Wang, Jiahao Xun, Minjie Hong, Jieming Zhu, Tao Jin, Wang Lin, Haoyuan Li, Linjun Li, Yan Xia, Zhou Zhao, et al. 2024. Eager: Two-stream generative recommender with behavior-semantic collaboration. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3245–3254.

[39] Yejing Wang, Shengyu Zhou, Jinyu Lu, Ziwei Liu, Langming Liu, Maolin Wang, Wenlin Zhang, Feng Li, Wenbo Su, Pengjie Wang, et al. 2026. Nezha: A zero-sacrifice and hyperspeed decoding architecture for generative recommendations. In *Proceedings of the ACM Web Conference 2026*. 8073–8082.

[40] Zesheng Wang, Longfei Xu, Weidong Deng, Huimin Yan, Kaikui Liu, and Xiangxiang Chu. 2026. IntRR: A Framework for Integrating SID Redistribution and Length Reduction. *arXiv preprint arXiv:2602.20704* (2026).

[41] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453* (2023).

[42] Liu Yang, Fabian Paischer, Kaveh Hassani, Jiacheng Li, Shuai Shao, Zhang Gabriel Li, Yun He, Xue Feng, Nima Noorshams, Sem Park, et al. 2024. Unifying generative and dense retrieval for sequential recommendation. *arXiv preprint arXiv:2411.18814* (2024).

[43] Yuhao Yang, Zhi Ji, Zhaopeng Li, Yi Li, Zhonglin Mo, Yue Ding, Kai Chen, Zijian Zhang, Jie Li, Shuanglong Li, et al. 2025. Sparse meets dense: Unified generative recommendations with cascaded sparse-dense representations. *arXiv preprint arXiv:2503.02453* (2025).

[44] Jiaqi Zhai, Lucy Liao, Xing Liu, Yueming Wang, Rui Li, Xuan Cao, Leon Gao, Zhaojie Gong, Fangda Gu, Michael He, et al. 2024. Actions speak louder than words: Trillion-parameter sequential transducers for generative recommendations. *arXiv preprint arXiv:2402.17152* (2024).

[45] Zhaoqi Zhang, Haolei Pei, Jun Guo, Tianyu Wang, Yufei Feng, Hui Sun, Shaowei Liu, and Aixin Sun. 2026. Onetrans: Unified feature interaction and sequence modeling with one transformer in industrial recommender. In *Proceedings of the ACM Web Conference 2026*. 8162–8170.

[46] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* 1, 2 (2023), 1–124.

[47] Zhe Zhao, Tao Liu, Shen Li, Bofang Li, and Xiaoyong Du. 2017. Ngram2vec: Learning improved word representations from ngram co-occurrence statistics. In *Proceedings of the 2017 conference on empirical methods in natural language processing*. 244–253.

[48] Bowen Zheng, Yupeng Hou, Hongyu Lu, Yu Chen, Wayne Xin Zhao, Ming Chen, and Ji-Rong Wen. 2024. Adapting large language models by integrating collaborative semantics for recommendation. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1435–1448.

[49] Guorui Zhou, Hengrui Hu, Hongtao Cheng, Huanjie Wang, Jiaxin Deng, Jinghao Zhang, Kuo Cai, Lejian Ren, Lu Ren, Liao Yu, et al. 2025. Onerec-v2 technical report. *arXiv preprint arXiv:2508.20900* (2025).

[50] Jieming Zhu, Mengqun Jin, Qijiong Liu, Zexuan Qiu, Zhenhua Dong, and Xiu Li. 2024. Cost: Contrastive quantization based semantic tokenization for generative recommendation. In *Proceedings of the 18th ACM Conference on Recommender Systems*. 969–974.

[51] Yanyan Zou, Junbo Qi, Lunsong Huang, Yu Li, Kewei Xu, Jiabao Gao, Binglei Zhao, Xuanhua Yang, Sulong Xu, and Shengjie Li. 2026. GenRec: A Preference-Oriented Generative Framework for Large-Scale Recommendation. *arXiv preprint arXiv:2604.14878* (2026).

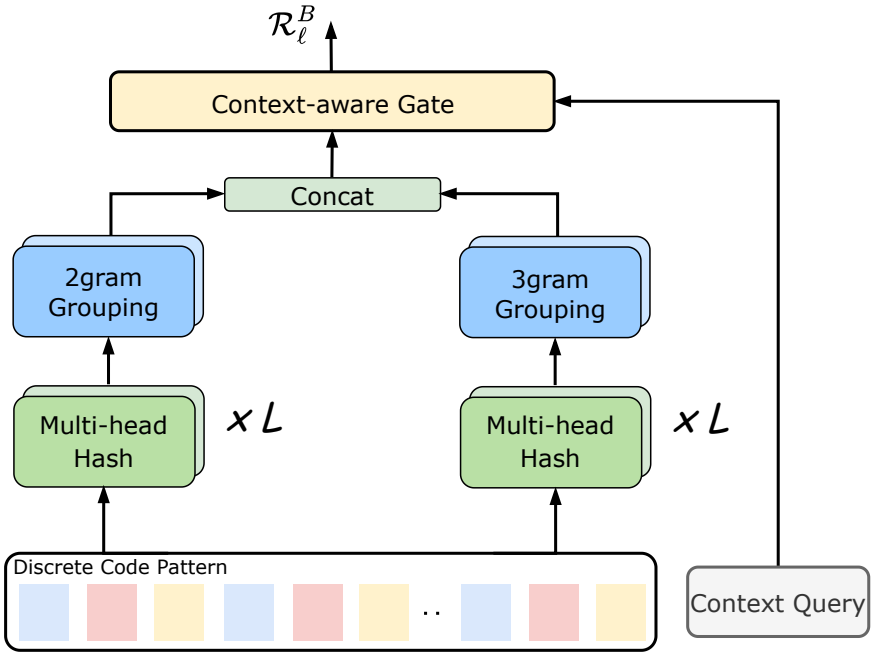


Figure 4: The detailed framework of the general Engram module. A discrete code sequence is converted into suffix N-gram keys of multiple orders, retrieved from multi-head sparse hash tables, and modulated by a context-aware gate before being returned as conditional memory evidence.

A Supplement to Method

Due to the main text length limitation, this section complements Section 3 with implementation details that are not fully expanded in the main text.

Roadmap. The supplement focuses on two parts. Sections A.1–A.3 provide the implementation details of Dual-level Engram Memory, including hash-table addressing, context-aware gating, and memory-specific discrete units. Section A.4 details Memory-restoring Prediction Head, including candidate-specific memory restoration, catalog-valid decoding, and architecture-specific decoding states.

A.1 General Engram

The main text defines the Engram read operator $\mathcal{R}_\ell(\boldsymbol{q}, \boldsymbol{p})$. Here we only expand how an abstract suffix key is mapped into sparse tables. We use K for the number of hash heads in a memory instance, d_m for the concatenated Engram address dimension, and d for the LLM hidden size used by the SID-token embeddings in Equation (2). Intra-item and inter-item memories use separate tables, even though they share the same addressing form.

Given $\boldsymbol{p} = (p_1, \ldots, p_T)$, the level-$\ell$ memory extracts ending N-gram keys from the lookup pattern. In intra-item memory, each p_t is a SID code. In inter-item memory, each p_t is a SID prefix $\boldsymbol{c}_n^{\leq \ell}$ and is treated as one discrete unit. For order o, the ending key is

$$g_o(\boldsymbol{p}) = (p_{\max(1,T-o+1)}, \ldots, p_T). \tag{13}$$

The set of N-gram orders used at level ℓ is denoted as O_ℓ. Each key is mapped to K sparse tables by deterministic hash functions, and the retrieved vectors are concatenated into an address:

$$\boldsymbol{a}_{\ell,o}(\boldsymbol{p}) = \Big\Vert_{k=1}^{K} \boldsymbol{M}_{\ell,o,k}\left[\phi_{\ell,o,k}\left(g_o(\boldsymbol{p})\right)\right], \tag{14}$$

where $\Vert$ denotes vector concatenation, $\phi_{\ell,o,k}(\cdot)$ is the k-th hash function, and $\boldsymbol{M}_{\ell,o,k} \in \mathbb{R}^{H_{\ell,o,k} \times d_m/K}$ is the corresponding learnable sparse table with $H_{\ell,o,k}$ buckets. The address $\boldsymbol{a}_{\ell,o}(\boldsymbol{p}) \in \mathbb{R}^{d_m}$ depends only on the discrete pattern and is therefore filtered by the context-aware gate used in Equation (5):

$$\lambda_{\ell,o} = \sigma\left(\frac{\left(\boldsymbol{W}_{Q,\ell}\boldsymbol{q}\right)^\top \left(\boldsymbol{W}_{K,\ell,o}\boldsymbol{a}_{\ell,o}(\boldsymbol{p})\right)}{\sqrt{d}}\right). \tag{15}$$

The final read is the same operator as Equation (5), written here with the explicit hash-table address:

$$\mathcal{R}_\ell(\boldsymbol{q}, \boldsymbol{p}) = \mathrm{LN}\left(\sum_{o \in O_\ell} \lambda_{\ell,o}\boldsymbol{W}_{V,\ell,o}\boldsymbol{a}_{\ell,o}(\boldsymbol{p})\right). \tag{16}$$

Hash table capacities and the concrete hash function are specified in Section B.2.2.

A.2 Intra-item Engram

Section 3.3 already gives the intra-item read and aggregation equations. The only implementation convention needed here is the discrete lookup pattern. Since the first code has no previous prefix, intra-item patterns start from the second code:

$$\boldsymbol{p}_{n,\ell}^{S} = (\boldsymbol{c}_n^{<\ell}, c_n^{\ell}), \quad \ell = 2, \ldots, L, \tag{17}$$

The encoder-side read uses s_n^0 as the query, as in Equation (6). The adapter $(W_{C,\ell}^S, b_{C,\ell}^S)$ in that equation only appears during representation construction; during decoding, Section A.4 directly queries the same intra-item table with $(c_{N+1}^{<\ell}, x)$.

A.3 Inter-item Engram

Section 3.3 defines the inter-item memory used by the representation constructor. For implementation, the important point is that one inter-item unit is an encoded SID prefix, not a raw item ID. At each level ℓ, these units form

$$C^{\le \ell} = \left(c_1^{\le \ell}, c_2^{\le \ell}, \ldots, c_N^{\le \ell}\right), \quad \ell = 1, \ldots, L. \tag{18}$$

For a sequence $\boldsymbol{x} = (x_1, \ldots, x_T)$, define its recent suffix ending at position t as

$$\mathrm{Suffix}_o(\boldsymbol{x}, t) = (x_{\max(1,t-o+1)}, \ldots, x_t). \tag{19}$$

The representation-side transition pattern is

$$p_{n,\ell}^T = \mathrm{Suffix}_{\tau_\ell}\left(C^{\le \ell}, n\right), \tag{20}$$

where $\tau_\ell \ge \max O_\ell$ ensures that all N-gram orders can be extracted. Since transitions depend on more than the current item alone, we compute the transition-aware query from a local window $\mathcal{H}_n = \{a \mid \max(1, n-w+1) \le a \le n\}$ of MM-guided item contexts:

$$q_{n,\ell}^T = \sum_{a \in \mathcal{H}_n} \pi_{a,\ell} s_a^0, \quad \pi_{a,\ell} = \mathrm{softmax}_{a \in \mathcal{H}_n}\left(\frac{\left(W_{Q,\ell}^{\mathrm{P}} s_n^0\right)^\top \left(W_{K,\ell}^{\mathrm{P}} s_a^0\right)}{\sqrt{d}}\right). \tag{21}$$

Here $\pi_{a,\ell}$ weights item position a's contribution to the level-ℓ transition query, and $W_{Q,\ell}^{\mathrm{P}}, W_{K,\ell}^{\mathrm{P}}$ are level-specific projections.

A.4 Memory-restoring Prediction Head

The prediction head uses the same Engram tables during decoding. After the generator consumes R^I, let ζ_ℓ denote the architecture-specific state supplied to the prediction head at SID layer ℓ. In the normal GR architecture, $\zeta_\ell = h_u$ for all layers; in the NEZHA-style architecture, ζ_ℓ is replaced by the layer-specific state defined in Section B.2.3. This notation keeps the memory-restoring equations shared by both architectures. When predicting the ℓ-th code, the current generated prefix is $c_{N+1}^{<\ell}$ and a candidate code is x. The corresponding level-ℓ candidate prefix is

$$c_{N+1}^{\le \ell}(x) = (c_{N+1}^1, \ldots, c_{N+1}^{\ell-1}, x), \quad c_{N+1}^{\le 1}(x) = (x). \tag{22}$$

Candidate-specific Memory. For intra-item decoding, no intra-item memory is available at level 1. For $\ell > 1$, the candidate code is retrieved together with the generated prefix:

$$\bar{\eta}_{N+1,\ell}^S(x) = \begin{cases} \mathbf{0}, & \ell = 1, \\ \mathcal{R}_\ell\left(\zeta_\ell, (c_{N+1}^{<\ell}, x)\right), & \ell > 1. \end{cases} \tag{23}$$

For inter-item decoding, we append the candidate prefix to the historical prefix sequence:

$$C_+^{\le \ell}(x) = [C^{\le \ell}; c_{N+1}^{\le \ell}(x)]. \tag{24}$$

The candidate transition pattern is $p_{N+1,\ell}^T(x) = \mathrm{Suffix}_{\tau_\ell}(C_+^{\le \ell}(x), N+1)$. Since the target item is unknown, the transition query is computed from the architecture-specific decoding state and the recent historical item contexts:

$$\rho_{a,\ell} = \mathrm{softmax}_{a \in \mathcal{H}_N}\left(\frac{\left(W_{Q,\ell}^{\mathrm{D}} \zeta_\ell\right)^\top \left(W_{K,\ell}^{\mathrm{D}} s_a^0\right)}{\sqrt{d}}\right), \quad q_{u,\ell}^{\mathrm{D}} = \sum_{a \in \mathcal{H}_N} \rho_{a,\ell} s_a^0. \tag{25}$$

The inter-item memory for candidate x is

$$\eta_{N+1,\ell}^T(x) = \mathcal{R}_\ell\left(q_{u,\ell}^{\mathrm{D}}, p_{N+1,\ell}^T(x)\right). \tag{26}$$

These memories check two conditions for candidate x: whether it is compatible with the generated SID prefix, and whether it is compatible with the recent transition context.

Memory Fusion and Candidate Scoring. The memory term used in Equations (8) and (9) is

$$\mu_\ell(x) = \begin{cases} W_{T,1}\eta_{N+1,1}^T(x), & \ell = 1, \\ W_{S,\ell}\bar{\eta}_{N+1,\ell}^S(x) + W_{T,\ell}\eta_{N+1,\ell}^T(x), & \ell > 1. \end{cases} \tag{27}$$

The candidate embedding e_x in Equation (10) is the SID-token embedding of candidate code x at layer ℓ, defined in Equation (2). The logit therefore combines three signals: the architecture-specific decoding state ζ_ℓ, the candidate code embedding e_x, and the memory evidence $\mu_\ell(x)$. For the main-text equations, ζ_ℓ reduces to h_u unless the NEZHA-style variant is used.

Catalog-valid Prefix Decoding. Let $\mathcal{T}$ denote the prefix tree built from all item SIDs in the catalog. At level ℓ, the valid candidate set $\mathcal{V}_\ell(c_{N+1}^{<\ell})$ contains the children of the current prefix in $\mathcal{T}$. The probability in Equation (11) is normalized only over this set. During beam search, each partial SID keeps its accumulated score

$$s_\ell = \sum_{t=1}^{\ell} \log P\left(c_{N+1}^t \middle| R^I, c_{N+1}^{<t}; \Theta\right). \tag{28}$$

Completed SIDs are mapped back to catalog items. If multiple valid SIDs map to the same item due to SID collisions, the item is ranked by its best valid SID score.

B Experimental Settings

B.1 Datasets Statistics

In this section, we present the detailed statistics of the selected public datasets, *i.e.,* Yelp, Amazon Industrial, and Amazon Instrument. For data preprocessing, we follow previous sequential recommendation and generative retrieval settings [13, 32]. The statistics after preprocessing are presented in Table 4.

Table 4: The statistics of datasets.

Dataset	# Users	# Items	Sparsity	Avg.length
Yelp	15,720	11,383	99.89%	12.23
Industrial	27,190	28,461	99.92%	6.56
Instrument	40,644	30,676	99.97%	8.01

B.2 Detailed Implementation

The hardware used in all experiments is an AMD EPYC 9745 platform with 2 NVIDIA RTX PRO 6000 (Blackwell, 96GB) GPUs, while the basic software requirements are Python 3.11 and PyTorch 2.10.

Next, we detail the implementation of the quantization, representation, and generation stages for the adopted baselines. For ComeIR, the multimodal item embedding $\boldsymbol{m}_n$ is the cached input used by the quantizer to produce SIDs, and we do not introduce an additional external-input branch $\boldsymbol{b}_n$ in the representation constructor.

B.2.1 Quantization. We adopt RQ-VAE [19] to obtain hierarchical SIDs. To keep the notation consistent with the main text, the multimodal item embedding of item v_n is denoted as $\boldsymbol{m}_n$. The encoder maps it into a latent vector $\boldsymbol{z}_n = \mathrm{Enc}_Q(\boldsymbol{m}_n)$. Starting from $\boldsymbol{r}_{n,0} = \boldsymbol{z}_n$, the ℓ-th code is assigned by residual quantization:

$$c_n^\ell = \arg\min_{j\in\{1,\dots,C_\ell\}} \left\|\boldsymbol{r}_{n,\ell-1} - \boldsymbol{b}_{\ell,j}\right\|_2^2,$$
$$\boldsymbol{r}_{n,\ell} = \boldsymbol{r}_{n,\ell-1} - \boldsymbol{b}_{\ell,c_n^\ell}, \quad \hat{\boldsymbol{z}}_n = \sum_{\ell=1}^{L} \boldsymbol{b}_{\ell,c_n^\ell}. \tag{29}$$

Here C_ℓ is the size of the ℓ-th codebook, $\boldsymbol{b}_{\ell,j}$ is its j-th vector, Enc_Q and Dec_Q are the quantizer encoder and decoder, and $\boldsymbol{r}_{n,\ell}$ is the residual after the first ℓ code assignments. After quantization, we use $\boldsymbol{e}^B_{c_n^\ell} = \boldsymbol{b}_{\ell,c_n^\ell}$ as the frozen codebook embedding in Equation (2). The quantizer is trained with reconstruction and residual commitment losses:

$$\mathcal{L}_Q = \left\|\boldsymbol{m}_n - \mathrm{Dec}_Q\left(\hat{\boldsymbol{z}}_n\right)\right\|_2^2 + \sum_{\ell=1}^{L}\left(\left\|\mathrm{sg}\left(\boldsymbol{r}_{n,\ell-1}\right) - \boldsymbol{b}_{\ell,c_n^\ell}\right\|_2^2 + \beta_Q\left\|\boldsymbol{r}_{n,\ell-1} - \mathrm{sg}\left(\boldsymbol{b}_{\ell,c_n^\ell}\right)\right\|_2^2\right). \tag{30}$$

where $\mathrm{sg}(\cdot)$ denotes stop-gradient and β_Q is the commitment weight. Following common RQ-VAE settings, we set $\beta_Q = 1.0$. In our main experiments, the SID length is $L = 3$ and each codebook contains 128 codes, yielding the three-layer SID space used by the representation and generation modules.

RQ-KMeans. Besides RQ-VAE, the main experiments also evaluate an RQ-KMeans quantizer. RQ-KMeans follows the same residual assignment logic as Equation (29), but the codebooks are obtained by iterative KMeans clustering over residual vectors rather than by optimizing an encoder-decoder reconstruction objective. Specifically, at layer ℓ, KMeans is fitted on the current residuals $\{\boldsymbol{r}_{n,\ell-1}\}$, the cluster index becomes c_n^ℓ, and the selected centroid is subtracted to form $\boldsymbol{r}_{n,\ell}$. After all L layers, the resulting SID still has the form $\boldsymbol{c}_n = (c_n^1, \dots, c_n^L)$ and is consumed by the same representation and generation modules. Thus, changing RQ-VAE to RQ-KMeans only changes the quantization stage; all Engram memories, token merge, and prediction-head definitions remain unchanged.

B.2.2 Representation. This subsection specifies how the sparse parameters used in the representation module are computed. Let C be the codebook size per SID layer; in our main experiments $L = 3$ and $C = 128$. We denote the maximum allowed bucket count of a single hash table by $H_{\max} = 20{,}000{,}000$ and cap the integer encoding domain of one discrete unit by $D_{\max} = 2{,}097{,}152$. The Engram address dimension is $d_m = 256$. Thus each intra-item head has dimension $d_m/K_\mathrm{S} = 128$ with $K_\mathrm{S} = 2$, and each inter-item head has dimension $d_m/K_\mathrm{T} = 64$ with $K_\mathrm{T} = 4$. The main-text statement "base 128 for intra-item and base 16 for inter-item" refers to the base used before applying the memory-specific scale values s_S and s_T.

Intra-item Engram Table Setting. For intra-item memory, a conditional pattern at level ℓ is $\boldsymbol{p}^S_{n,\ell} = (\boldsymbol{c}_n^{<\ell}, c_n^\ell)$ as defined in Equation (17). The pattern contains ℓ SID codes, so its exact discrete domain is C^ℓ; in implementation we use the encoded unit domain

$$D^S_\ell = \min(C^\ell, D_{\max}). \tag{31}$$

The intra-item memory uses the order set $\boldsymbol{O}^S_\ell = \{1, \dots, O^S_\ell\}$ with $O^S_\ell = \min(\ell, 3)$. Given an intra scale $s_\mathrm{S} > 0$, we first define the scaled intra base as

$$B^S(s_\mathrm{S}) = Cs_\mathrm{S} = 128 s_\mathrm{S}. \tag{32}$$

The target bucket count before assigning hash heads is

$$\Gamma^S_{\ell,o}(s_\mathrm{S}) = \max\left(2, \min\left(\left\lfloor B^S(s_\mathrm{S})^o \right\rfloor, C^o, H_{\max}\right)\right), \quad o \in \boldsymbol{O}^S_\ell. \tag{33}$$

This equation is the concrete meaning of scaling up intra-item memory. Increasing s_S enlarges the effective code base $B^S(s_\mathrm{S})$ before the order-o power is taken. For example, with $C = 128$, $s_\mathrm{S} = 0.5$ gives target bases 64^o, while $s_\mathrm{S} = 1.0$ reaches the collision-free domain 128^o for the used intra orders unless the global cap $H_{\max}$ is active. Values above 1.0 are clipped by the exact intra domain C^o, because intra patterns are formed within one SID and the full code-combination domain is already enumerable.

Each hash head uses a distinct prime bucket size near this target. Formally, for $k = 1, \dots, K_\mathrm{S}$, $H^S_{\ell,o,k}$ is selected from unused primes around $\Gamma^S_{\ell,o}(s_\mathrm{S})$ so that different heads use different moduli. The sparse parameters contributed by intra hash tables are therefore

$$P_\mathrm{S}(s_\mathrm{S}) = \frac{d_m}{K_\mathrm{S}} \sum_{\ell=2}^{L} \sum_{o\in\boldsymbol{O}^S_\ell} \sum_{k=1}^{K_\mathrm{S}} H^S_{\ell,o,k}. \tag{34}$$

The scaling figures in Section 4.4 report the resulting sparse parameter count P_S, not merely the raw scale value s_S.

Inter-item Engram Table Setting. For inter-item memory, one discrete unit at SID level ℓ is the prefix $\boldsymbol{c}_n^{\le\ell} = (c_n^1, \dots, c_n^\ell)$, encoded as one integer. Thus its exact prefix domain is C^ℓ and the implementation unit domain is

$$D^T_\ell = \min(C^\ell, D_{\max}). \tag{35}$$

We use $\boldsymbol{O}^T_\ell = \{1, \dots, O^T_\ell\}$ with $(O^T_1, O^T_2, O^T_3) = (3, 2, 1)$. This allocates longer transition contexts to coarse prefixes and shorter contexts to deeper, more specific prefixes.

Inter-item capacity is scaled with a fixed base of 16. Since an inter-item unit at level ℓ is an encoded prefix rather than a single SID code, we use the level-wise scaled inter base

$$B^T_\ell(s_\mathrm{T}) = (16 s_\mathrm{T})^\ell. \tag{36}$$

This is the concrete meaning of the "base 16" inter-item setting in Section 4.1. Increasing s_T enlarges the base at every level before the N-gram power is taken. For example, the default $s_\mathrm{T} = 2.0$ gives $B^T_1 = 32$, $B^T_2 = 1024$, and $B^T_3 = 32768$.

The target bucket count is then

$$\Gamma^T_{\ell,o}(s_\mathrm{T}) = \max\left(2, \min\left(\left\lfloor B^T_\ell(s_\mathrm{T})^o \right\rfloor, (C^\ell)^o, H_{\max}\right)\right), \quad o \in \boldsymbol{O}^T_\ell. \tag{37}$$

The term $(C^\ell)^o$ is the full prefix-transition domain for an order-o inter key at level ℓ, and $H_{\max}$ is the global per-table cap.

Algorithm 1 Training procedure of ComeIR

Require: User sequence $\mathcal{S}_u$, target SID $\boldsymbol{c}_{N+1}$, architecture type $A \in \{\text{Normal}, \text{NEZHA}\}$, multimodal item embeddings, SID codebooks

Build SID-token embeddings by Equation (2)
Compute MM-guided item contexts by Equation (4)
Retrieve intra-item and inter-item memories by Equations (6) and (??)
Set $\bar{\boldsymbol{\eta}}^S_{n,1} = \mathbf{0}$ and $\bar{\boldsymbol{\eta}}^S_{n,\ell} = \boldsymbol{\eta}^S_{n,\ell}$ for $\ell > 1$
Merge SID evidence into item-level inputs $\boldsymbol{R}^I$ by Equation (7)
if A = Normal **then**
    Encode $\boldsymbol{R}^I$ with the backbone to obtain $\boldsymbol{h}_u$
    Set $\boldsymbol{\zeta}_\ell = \boldsymbol{h}_u$ for all ℓ
else
    Append SID-layer placeholders and encode by Equation (40)
    Initialize $\boldsymbol{h}^1_u = \boldsymbol{h}_u$
end if
for $\ell = 1$ to L **do**
    Use the ground-truth prefix $\boldsymbol{c}^{<\ell}_{N+1}$ as the decoding prefix
    if A = NEZHA **then**
        Compute $\boldsymbol{\zeta}_\ell = \boldsymbol{\xi}_\ell$ from $\boldsymbol{h}^\ell_u$ and the ℓ-th placeholder state by Equation (41)
    end if
    Restore candidate-specific memories with state $\boldsymbol{\zeta}_\ell$ by Equations (23) and (26)
    Compute the valid-prefix probability by Equation (11)
    if A = NEZHA and $\ell < L$ **then**
        Update $\boldsymbol{h}^{\ell+1}_u$ with the ground-truth code c^ℓ_{N+1} by Equation (42)
    end if
end for
Optimize the token-level cross-entropy in Equation (12)

Algorithm 2 Inference procedure of ComeIR

Require: User sequence $\mathcal{S}_u$, architecture type $A \in \{\text{Normal}, \text{NEZHA}\}$, catalog prefix tree $\mathcal{T}$, beam size B_{beam}

Construct $\boldsymbol{R}^I$ from the historical items
if A = Normal **then**
    Encode $\boldsymbol{R}^I$ to obtain $\boldsymbol{h}_u$ and set $\boldsymbol{\zeta}_\ell = \boldsymbol{h}_u$ for all ℓ
    Initialize the beam with the empty prefix and score 0
else
    Append SID-layer placeholders and obtain $\boldsymbol{h}_u$ and placeholder states by Equation (40)
    Initialize the beam with the empty prefix, score 0, and recurrent state $\boldsymbol{h}^1_u = \boldsymbol{h}_u$
end if
for $\ell = 1$ to L **do**
    for each partial prefix in the beam **do**
        Enumerate catalog-valid codes from $\mathcal{T}$
        if A = NEZHA **then**
            Compute $\boldsymbol{\zeta}_\ell = \boldsymbol{\xi}_\ell$ from the beam recurrent state and the ℓ-th placeholder state
        end if
        Restore candidate-specific memories with state $\boldsymbol{\zeta}_\ell$ and compute layer-wise log-probabilities
        if A = NEZHA **then**
            Attach the updated recurrent state from Equation (42) to each expanded beam hypothesis
        end if
    end for
    Keep the top-B_{beam} partial SIDs by accumulated log-probability
end for
Map completed valid SIDs to catalog items and rank them by accumulated score

As in the intra-item memory, each inter head receives a distinct prime bucket size $H^T_{\ell,o,k}$ near $\Gamma^T_{\ell,o}(s_{\mathrm{T}})$. The sparse parameters contributed by inter hash tables are

$$P_{\mathrm{T}}(s_{\mathrm{T}}) = \frac{d_m}{K_{\mathrm{T}}} \sum_{\ell=1}^{L} \sum_{o \in O^T_\ell} \sum_{k=1}^{K_{\mathrm{T}}} H^T_{\ell,o,k}. \tag{38}$$

Therefore, scaling up the inter-item Engram by s_{T} increases the level base before the order-o power and then maps the resulting targets into prime-sized sparse hash tables. The actual plotted x-axis in Section 4.4 is the computed parameter count $P_{\mathrm{T}}(s_{\mathrm{T}})$.

Hash Function. For a memory type $X \in \{S, T\}$, a key $(x_1, \ldots, x_o)$ is mapped by deterministic multi-head hashing.

$$\phi^X_{\ell,o,k}(x_1, \ldots, x_o) = \left(\bigoplus_{a=1}^{o} x_a m^X_{a,k} \right) \bmod H^X_{\ell,o,k}, \tag{39}$$

where $\oplus$ denotes bitwise XOR and $m^X_{a,k}$ is an odd deterministic multiplier generated from the random seed. This construction keeps the memory sparse and scalable while allowing table capacity to grow with either the exact intra-SID code-combination domain or the level-wise inter-item prefix-transition domain.

B.2.3 Generation. We instantiate ComeIR with two generation architectures: a normal GR architecture and a NEZHA-style architecture [39]. Both use the same quantization and representation modules, and differ only in how the generator exposes hidden states to the prediction head.

Normal GR.. In the normal architecture, the backbone directly consumes $\boldsymbol{R}^I$ and returns hidden states $\boldsymbol{H} = \mathrm{LLM}(\boldsymbol{R}^I)$. The last valid hidden state is used as $\boldsymbol{h}_u$, and the state supplied to the prediction head is $\boldsymbol{\zeta}_\ell = \boldsymbol{h}_u$ for every SID layer. The Memory-restoring Prediction Head in Section 3.5 then performs layer-wise SID generation. This architecture is the most direct plug-in form of ComeIR, because it replaces the input representation while keeping the standard autoregressive GR decoding interface.

NEZHA Architecture. Following the NEZHA decoding architecture, we append one randomly initialized trainable placeholder for each SID layer after the item-level sequence. Since our experiments use $L = 3$, the input contains three placeholders $\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3 \in \mathbb{R}^d$:

$$\boldsymbol{H}^+ = \mathrm{LLM}\left([\boldsymbol{R}^I; \boldsymbol{p}_1; \boldsymbol{p}_2; \boldsymbol{p}_3] \right). \tag{40}$$

Let $\boldsymbol{h}_u$ be the hidden state of the last historical item and let $(\boldsymbol{h}_1, \boldsymbol{h}_2, \boldsymbol{h}_3)$ be the hidden states of the three placeholders. The prediction head

no longer reads only $\boldsymbol{h}_u$; instead, the ℓ-th SID layer uses a layer-specific state

$$\boldsymbol{\xi}_\ell = \boldsymbol{h}_u^\ell + \boldsymbol{h}_\ell, \quad \ell = 1, 2, 3, \tag{41}$$

where $\boldsymbol{h}_u^1 = \boldsymbol{h}_u$, and $\boldsymbol{h}_u^\ell$ denotes the recurrent user state before predicting the ℓ-th code. The state supplied to the prediction head is $\boldsymbol{\zeta}_\ell = \boldsymbol{\xi}_\ell$, while the same candidate-specific intra-item and inter-item memories are restored. After the ℓ-th code is generated, the recurrent user state is updated by a GRU transition:

$$\boldsymbol{h}_u^{\ell+1} = \mathrm{GRU}_\ell\left(\Delta_\ell\left(c_{N+1}^\ell\right), \boldsymbol{h}_u^\ell\right), \tag{42}$$

where $\Delta_\ell(c_{N+1}^\ell)$ denotes the transition input constructed from the generated code embedding and the next-level inter-item context. Teacher forcing provides c_{N+1}^ℓ during training, and beam search provides the selected candidate during inference. This design lets the placeholders provide layer-wise draft contexts, while the GRU carries generated-code information across SID layers.

Optimization Settings. Unless otherwise specified, both generation architectures are trained with bfloat16 precision, base learning rate 1×10^{-5}, weight decay 1×10^{-4}, per-device batch size 64, gradient accumulation steps 2, and maximum training steps 100,000. We use three SID layers with codebook sizes $(128, 128, 128)$, evaluate every 100 steps, and report the average over random seeds $\{42, 43, 44\}$.

B.3 Training and Inference Procedures

Algorithm 1 summarizes the training procedure of ComeIR. The quantization and representation steps are shared by the normal GR and NEZHA-style architectures: historical SIDs are transformed into SID-token embeddings, converted into MM-guided item contexts, enriched by intra-item and inter-item memories, and merged into item-level representations $\boldsymbol{R}^I$. The two architectures differ only after $\boldsymbol{R}^I$ is constructed. Normal GR feeds $\boldsymbol{R}^I$ to the backbone and uses the last valid user state $\boldsymbol{h}_u$ for every SID layer. NEZHA appends layer placeholders, obtains placeholder states, and supplies the layer-specific state $\boldsymbol{\zeta}_\ell = \boldsymbol{\xi}_\ell$ to the same prediction head. During training, both variants use the ground-truth SID prefix at each layer; NEZHA additionally updates its recurrent user state with the ground-truth code under teacher forcing.

Algorithm 2 describes the inference procedure. The representation constructor is still applied once to the historical sequence. Normal GR keeps a single user state throughout beam search. NEZHA keeps an additional recurrent state for each beam hypothesis, because each generated SID code changes the state used by later SID layers. At each layer, the prefix tree restricts the candidate set to catalog-valid continuations. For every partial prefix in the beam, the prediction head restores intra-item evidence from the candidate prefix and inter-item evidence from the appended historical transition pattern. The beam keeps the partial SIDs with the largest accumulated log-probabilities, and the final valid SIDs are mapped back to catalog items for ranking.

C Extra Experimental Results

C.1 Ablation Study

In this section, we present the ablation study on the other two datasets, *i.e.,* Industrial and Instrument, all under the same LLM backbone (Qwen3-0.6B) and Quantization (RQ-VAE). The results further validate that all of our designed modules are effective, which offer significant performance gains.

Table 5: Ablation results on Instrument dataset. *w/o* MM-Scoring replaces MM-guided Token Scoring with mean pooling, and *w/o* Mem. Merge replaces the memory-conditioned token merge with a linear layer. Other variants remove intra-item or inter-item memory from the encoding (E) or decoding (D) stage.

Variant	H@5	H@10	N@5	N@10
ComeIR	**0.0834**	**0.1021**	**0.0709**	**0.0772**
w/o MM-scoring	0.0802	0.1010	0.0677	0.0744
w/o **D**-intra	0.0810	0.0970	0.0693	0.0752
w/o **D**-inter	0.0792	0.0982	0.0675	0.0723
w/o Mem. Merge	0.0817	0.0995	0.0695	0.0748
w/o **E**-intra	0.0812	0.0996	0.0690	0.0711
w/o **E**-inter	0.0806	0.1002	0.0667	0.0718

Table 6: Ablation results on Industrial dataset. *w/o* MM-Scoring replaces MM-guided Token Scoring with mean pooling, and *w/o* Mem. Merge replaces the memory-conditioned token merge with a linear layer. Other variants remove intra-item or inter-item memory from the encoding (E) or decoding (D) stage.

Variant	H@5	H@10	N@5	N@10
ComeIR	**0.1031**	**0.1321**	**0.0792**	**0.0881**
w/o MM-scoring	0.0998	0.1298	0.0764	0.0847
w/o **D**-intra	0.0989	0.1258	0.0775	0.0846
w/o **D**-inter	0.0987	0.1280	0.0749	0.0828
w/o Mem. Merge	0.1013	0.1295	0.0770	0.0863
w/o **E**-intra	0.1008	0.1277	0.0779	0.0815
w/o **E**-inter	0.0985	0.1305	0.0736	0.0830

C.2 Details of Scalability Analysis

This section explains how the sparse-parameter axis in Figure 3 is obtained from the scale values. We vary one memory type at a time while keeping the other type at its default setting: $s_\mathrm{T} = 2.0$ for the intra-item analysis and $s_\mathrm{S} = 1.0$ for the inter-item analysis. The plotted x-axis is the table parameter count computed by Equations (34) and (38), rather than the raw scale value.

The effective bases induced by these scales are straightforward. For the intra-item Engram table, $B^S(s_\mathrm{S}) = 128 s_\mathrm{S}$, so

$$s_\mathrm{S} \in \{0.125, 0.25, 0.5, 0.75, 1.0\}$$

corresponds to bases $\{16, 32, 64, 96, 128\}$. The order-o target bucket count is $\lfloor B^S(s_\mathrm{S})^o \rfloor$, clipped by the exact intra domain C^o and $H_{\max}$, then expanded to $K_\mathrm{S} = 2$ prime-sized heads and multiplied by the per-head dimension 128 to obtain P_S. For the inter-item Engram table, the underlying transition base is $16 s_\mathrm{T}$, so

$$s_\mathrm{T} \in \{0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0\}$$

corresponds to bases $\{8, 16, 24, 32, 40, 48, 56, 64\}$. At SID level ℓ, this becomes $B_\ell^T(s_T) = (16s_T)^\ell$; for example, the default $s_T = 2.0$ gives level bases $\{32, 1024, 32768\}$. The order-o target is clipped by the prefix-transition domain $(C^\ell)^o$ and $H_{\max}$, then expanded to $K_T = 4$ prime-sized heads and multiplied by the per-head dimension 64 to obtain P_T.

The two memories exhibit different scaling behavior because their pattern spaces are different. The intra-item Engram table is relatively small: all patterns are bounded by the code combinations inside one SID, and base 128 already reaches the theoretical capacity of the used intra-item domain. As s_S increases from 0.125 to 1.0, H@5 first improves almost linearly and then saturates. The gain becomes weaker around $s_S = 0.75$, because the number of observed intra-item code combinations is limited and multi-head hashing has already removed most effective collisions; further capacity mainly creates unused or rarely used buckets.

The inter-item Engram table has a much larger combinatorial space, since its keys describe cross-item prefix transitions. In the tested range $s_T \in \{0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0\}$, the inter table grows from about 1.4M to 4.431B sparse parameters, and H@5 keeps improving with the allocated sparse capacity. This indicates that reducing hash collisions remains useful for inter-item transitions within this range. We further test larger scales, $s_T \in \{5.0, 6.0, 8.0\}$, where $s_T = 8.0$ reaches the theoretical base upper bound. These larger tables all lead to different degrees of H@5 degradation, and performance becomes worse once the scale exceeds 4.0. We attribute this to sparse over-allocation: as the table size grows exponentially, most buckets are never activated or only receive very few updates, making it difficult to learn reliable memory vectors. This observation is consistent with the U-shaped sparsity-allocation phenomenon reported by Engram [6], where excessive sparse memory relative to backbone parameters can hurt performance under a fixed allocation trade-off. In our setting, when $s_T \geq 5.0$, the inter-item table is already on the order of ten times larger than the Qwen3-0.6B backbone.