Recommender Systems

Lecture 4: Generative Recommendation

Kidist Amde Mekonnen   Dominykas Seputis

k.a.mekonnen@uva.nl   d.seputis@uva.nl

University of Amsterdam

June 5, 2026

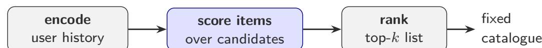
Lecture Outline

- **Recap: Classical Sequential Recommendation**
- **Motivation: From Cascades to Generation**
- **Generative Recommendation**
- **Item Tokenization: From Atomic ids to Semantic ids**
- **15 min coffee break**
- **Training and Decoding**
- **Limitations and Takeaways**

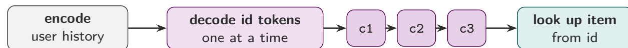
Where Are We in the Course?

- **Lecture 1:** Introduction to recommender systems.
- **Lecture 2:** Evaluation.
- **Lecture 3:** Sequential and LLM-based recommendation.
- **Lecture 4: Generative recommendation.**

Classical: score the catalogue



Generative: decode the identifier



The model generates an item id instead of scoring a fixed candidate set.

Key shift

The output space changes: instead of directly scoring catalogue items, the model generates **item-identifier tokens** that must map back to real items.

Main framing adapted from Rajput et al., *Recommender Systems with Generative Retrieval*, NeurIPS 2023.

Clarification: What Does “Generative” Mean Here?

“Generative” is used in different ways in RecSys, so the term alone is ambiguous.

Main focus of this lecture

Generate item identifiers

user history → item-id tokens
→ catalogue item

- The model generates a **tokenized item id**.
- The id is mapped back to an **existing item**.
- This is the main meaning of GenRec in this lecture.

TIGER: Rajput et al., *Recommender Systems with Generative Retrieval*, NeurIPS 2023.

Diffusion for embedding denoising

Example: DDRM

user/item embeddings
→ diffusion denoising
→ existing item candidates

- Diffusion is used to **denoise recommender embeddings**.
- Collaborative information guides the reverse denoising process.
- The output is grounded in the existing item pool; **no new item content is generated**.

DDRM: Zhao et al., SIGIR 2024.

Generative models for content generation

Example: DiFashion

user history + conditions
→ new fashion images
→ personalized outfit

- GANs, VAEs, and diffusion can generate **new item content**.
- DiFashion uses diffusion to generate **fashion images**.
- This is **new item-content generation**.

DiFashion: Xu et al., SIGIR 2024.

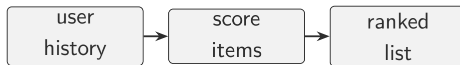
Recap: Classical Sequential Recommendation

Recap: Classical Sequential Recommendation

Task: next-item prediction

Given a chronologically ordered sequence of past user–item interactions, predict the next item.

Classical solution



$$\mathcal{H}_t = (i_1, i_2, \ldots, i_t)$$

$\mathcal{H}_t$: items the user clicked, watched, purchased, listened to, etc.

- Encode the user interaction history $\mathcal{H}_t$.
- Score each candidate catalogue item:

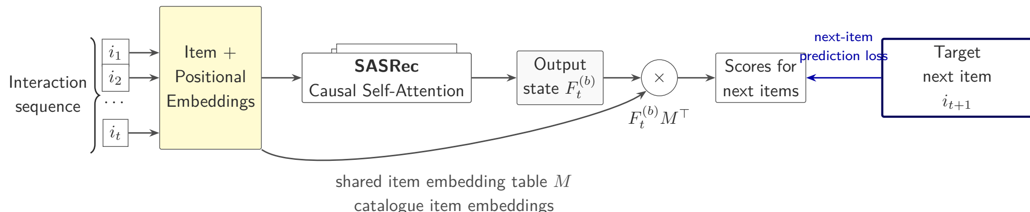
$$s(\mathcal{H}_t, i)$$

- Rank items by score.

Key point: classical sequential recommendation predicts the next item by **scoring catalogue items directly**.

SASRec, BERT4Rec, and GRU4Rec differ in how they encode $\mathcal{H}_t$, but they keep the same score-and-rank skeleton.

Classical Sequential Recommendation: Example with SASRec



- Each catalogue item has a unique **atomic item id** and a learned embedding.
- SASRec adds positional embeddings and applies **causal self-attention**.
- The output state $F_t^{(b)}$ is used to score candidate items.

$$r_{i,t} = F_t^{(b)} M_i^\top$$

A higher score means item i is more likely to be the next interaction.

SASRec is still a **score-and-rank** model: it encodes the user interaction history, then scores candidate **atomic item ids** directly.

Diagram adapted from Petrov & Macdonald, *Transformers for Sequential Recommendation*, ECIR 2024 (tutorial).
SASRec: Kang and McAuley, *Self-Attentive Sequential Recommendation*, ICDM 2018.

From Language-Model Ideas to GenRec

Language-model ideas entered recommendation in several ways, but most earlier uses still kept recommendation as scoring or text generation.

Architecture

Transformer encoders over interaction sequences

SASRec, ICDM 2018; BERT4Rec, CIKM 2019

Still scores catalogue items.

Representation

LMs encode item text, metadata, or reviews

RecFormer, 2023; item-text encoders

Still uses learned representations for ranking.

Task formulation

Recommendation tasks are written as prompts

P5, RecSys 2022; M6-Rec, 2022

Often generates text, ratings, or explanations.

GenRec shift

In Semantic id based generative recommendation, the generated sequence is not only an explanation or text response. It is the **item identifier** itself:

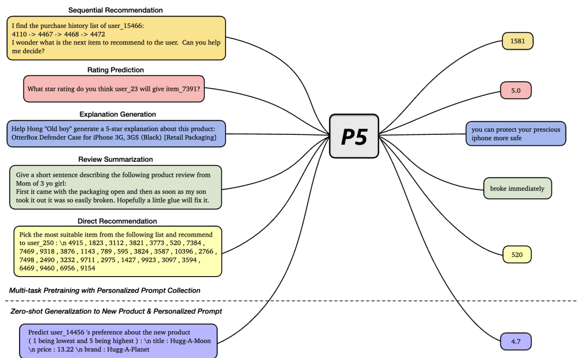
$$\text{user history} \longrightarrow \text{generated item id} \longrightarrow \text{catalogue item.}$$

GenRec examples: TIGER, NeurIPS 2023; OneRec, arXiv 2025.

Example: P5 Frames Recommendation as Language Generation

P5 idea

- Write recommendation tasks as **text-to-text** problems.
- Use prompts for different tasks:
  - sequential recommendation,
  - rating prediction,
  - explanation generation.
- This connects RecSys with sequence-to-sequence language modelling.

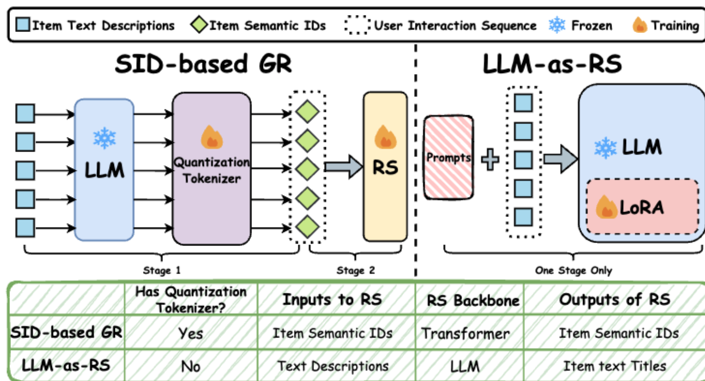


Source: Geng et al., *Recommendation as Language Processing*, RecSys 2022.

Bridge to GenRec

P5 casts recommendation as **text-to-text generation**; GenRec instead focuses on generating **catalogue-grounded item ids**.

Two Formulations of Generative Recommendation



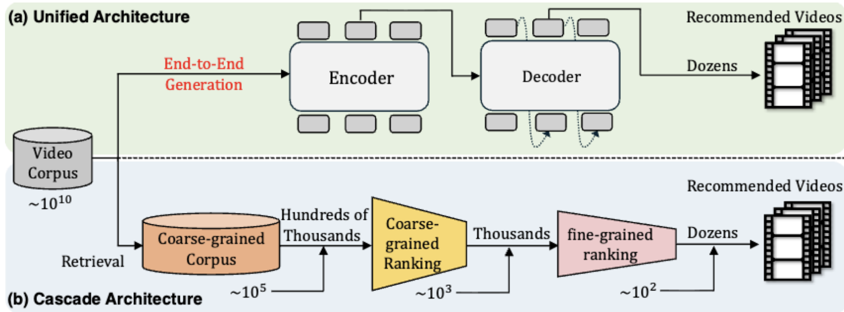
	Has Quantization Tokenizer?	Inputs to RS	RS Backbone	Outputs of RS
SID-based GR	Yes	Item Semantic IDs	Transformer	Item Semantic IDs
LLM-as-RS	No	Text Descriptions	LLM	Item text Titles

Source: Liu et al., *Understanding Generative Recommendation with Semantic IDs from a Model-Scaling View*, ICLR 2026 submission

This lecture focuses on **SID-based GenRec**: items are first mapped to **semantic ids**, and the recommender generates those ids. **LLM-as-RS** is a related formulation that uses item text directly.

Motivation: From Cascades to Generation

Motivation: From Cascades to End-to-End Generation



Deng et al., *OneRec*, arXiv 2025.

Multi-stage recommender

- Retrieve candidates → coarse ranking → fine ranking.
- Each stage narrows the item pool.
- Recall errors in early stages cannot be recovered by later rankers.

End-to-end generative recommender

- Ideally, a single model generates recommendation candidates directly.
- Candidate selection and ranking are learned jointly.
- This reduces handoffs between separate stages.

Motivation: Recommendation as Sequence Generation

The observation

- User behaviour is *already* a sequence:

$$i_1, i_2, \ldots, i_t \longrightarrow i_{t+1}.$$

- Next-item prediction is analogous to next-token prediction: use the past sequence to predict the next output.
- So instead of **scoring every candidate**, can the model **generate the next-item output** directly?

The shift

Classical encode history → **score catalogue** → rank
Generative encode history → **decode identifier** → look up item

The key challenge: generation needs a **token space**. Before the model can decode an item, we must decide **what an item identifier should look like**.

Motivation: Why Move Beyond Atomic Item ids?

The classical setup

- One **atomic item id** per item; a unique integer, one learned embedding each.
- Recommend by scoring candidate ids:

$$\mathcal{H}_t \;\rightarrow\; s(\mathcal{H}_t, i) \;\rightarrow\; \text{ranked items}$$

- The embedding table grows *linearly* with the catalogue.

Where it breaks

- **Scale:** output space = catalogue size; a softmax over millions of items.
- **Arbitrary:** `item_3487` says nothing about the item.
- **Cold start:** every new item needs a new id *and* a trained embedding.

So

Replace arbitrary ids with **generated identifiers** that are compact, valid, and carry structure.

Generative Recommendation

Generative Recommendation: Formal Setup

Notation. For each item i in the catalogue $\mathcal{I}$, a fixed-length identifier $\mathbf{z}_i = (z_{i,1}, \ldots, z_{i,L})$; a user history $\mathbf{x} = (x_1, \ldots, x_t)$ with $x_j \in \mathcal{I}$.

Autoregressive generation. The model decodes the next-item identifier one token at a time, each conditioned on the history and the tokens already generated:

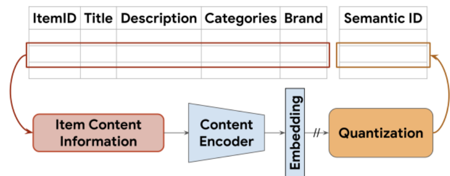
$$p_\theta(\mathbf{z}_i \mid \mathbf{x}) = \prod_{\ell=1}^{L} p_\theta(z_{i,\ell} \mid \mathbf{x},\, z_{i,<\ell}) \,.$$

Scoring view. The identifier's likelihood is a score for the item:

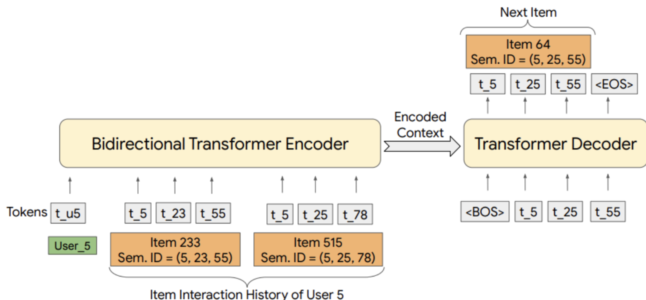
$$s_\theta(\mathbf{x}, i) = \log p_\theta(\mathbf{z}_i \mid \mathbf{x}).$$

Higher likelihood = more compatible with the user history. Recommendations are produced by **decoding valid item identifiers**, rather than by scoring a fixed candidate set directly.

Two Stages in Semantic ID Based Generative Recommendation



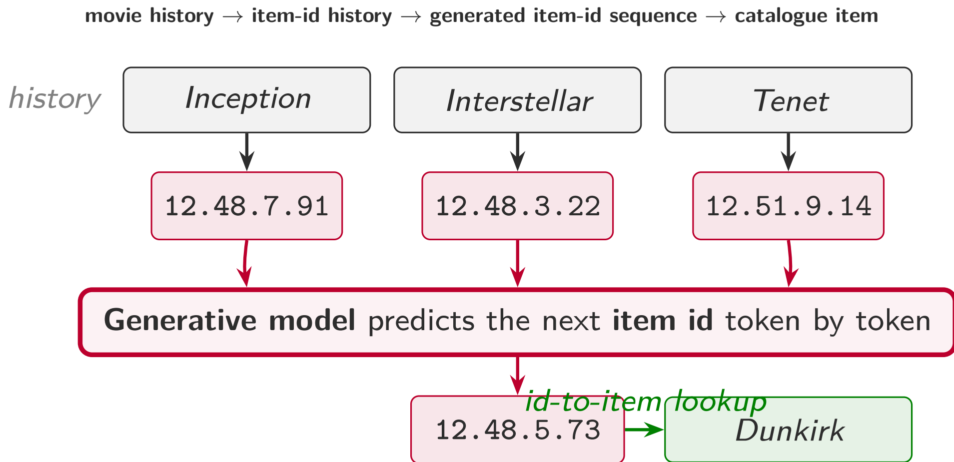
(a) Semantic ID generation for items using quantization of content embeddings.



(b) Transformer based encoder-decoder setup for building the sequence-to-sequence model used for generative retrieval.

Source: Rajput et al., *Recommender Systems with Generative Retrieval*, NeurIPS 2023.

Illustrative Example: Generating an Item Identifier

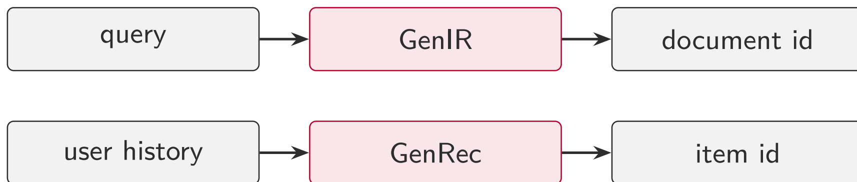


Shared prefix is illustrative: It suggests that the tokenizer has placed these movies in a similar coarse semantic region. That region could correspond to genre, director/style, metadata, or similar user behaviour, depending on what signals were used to build the item ids.

A Useful Parallel: Generative Information Retrieval

Earlier GenIR work, such as DSI, asked whether a model could generate a document identifier directly. **GenRec** applies the same idea to item identifiers.

Shared idea: generate a collection identifier instead of only scoring candidates directly.



Shared challenge: the generated id must be **valid**, **generatable**, and **grounded** in a real collection or catalogue.

Background: GENRE (De Cao et al., 2020); DSI (Tay et al., 2022); NCI (Wang et al., 2022); TIGER (Rajput et al., 2023).

Item Tokenization: From Atomic ids to Semantic ids

Recap: Tokenization Turns Data into Tokens

Generative models operate over token sequences. Before a model can generate, we must decide what the tokens are.

"The user watched Inception and Interstellar last night"

language tokenizer ↓ subword tokenizer, e.g., BPE (Byte-pair encoding), OpenAI `o200k_base` ↓;

`The` `user` `watched` `In` `ception` `and` `Inter` `stellar` `last` `night`

`976, 1825, 25301, 730, 1317, 326, 5605, 151024, 2174, 4856`

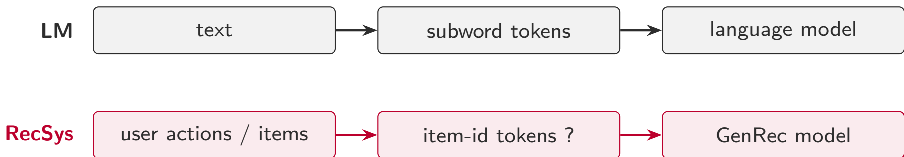
Illustrative token ids under OpenAI `o200k_base` tokenizer.

The model does not operate on raw words directly; it operates on token ids.

Bridge to GenRec

Language models generate **text tokens**. GenRec generate **item tokens**. So the key question shifts from **text tokenization** to **item tokenization**: how do we turn catalogue items into tokens the model can generate?

Why Item Tokenization Is Harder in GenRec



Text tokenization

- **Reusable subwords**
  e.g., *Interstellar* → `Inter` + `stellar`
- **Fixed vocabulary**
  often 30k–100k+ tokens; same token set covers many sentences
- **Dense supervision**
  common tokens appear in many contexts
- **Built into LMs**
  tokenizer is part of the pretrained model

Core question: Can we design item tokens that are **compact**, **valid**, **learnable**, and **structured**?

Item/action tokenization

- **Large catalogues**
  millions of possible items
- **Sparse interactions**
  few observations per user–item pair
- **Long-tail items**
  many items have little feedback
- **No natural subwords**
  `item_3487` has no reusable structure
- **Validity constraint**
  generated ids must map to real catalogue items

What Should the Item Identifier Look Like?

Every item needs an identifier. This choice defines the **output space** the generative recommender learns to produce.

A useful item id should be

- **compact**: short enough to generate efficiently.
- **grounded**: maps back to a real catalogue item.
- **learnable**: predictable from user histories.
- **structured**: related items can share parts of the code.

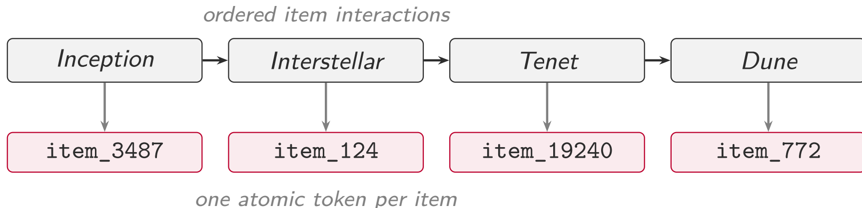
Three design choices

1. **Atomic ids**: one unique token per item.
2. **Textual ids**: use item text or metadata directly.
3. **Semantic ids**: short reusable token sequences.

The rest of this section compares these choices and asks which one makes item-id generation easier.

Background: Rajput et al., NeurIPS 2023; Hou et al., 2025.

Item Tokenization: One Atomic Token per Item?



Notice: four related films get four unrelated atomic tokens. The ids are compact and easy to look up, but they provide no reusable structure across items.

Why this is attractive

- Simple lookup.
- Short generated sequence.
- Direct mapping back to items.

Why this is weak

- Vocabulary grows with catalogue size: 10^6 items need 10^6 item tokens.
- Tokens are arbitrary identifiers.
- Similar items share no token structure.
- New items need new tokens and embeddings.

Item Tokenization: Use the Full Description as the id?

The other extreme: make the identifier the item's full text or metadata.

> *"A visually ambitious science-fiction film about memory, identity, artificial intelligence, and humanity's future, featuring a dark cyberpunk cityscape and a slow-burning mystery . . . "*

Why this is attractive

- Meaningful representation.
- Uses existing language tokens.
- Can exploit text and metadata.

Why this is weak

- Sequences become very long.
- Expensive to model and generate.
- Hard to constrain generation: many generated texts may not exactly match a catalogue item.
- May not map uniquely to one item.

Question: can we represent each item with a few reusable tokens: shorter than full text, but more structured than one atomic id?

This motivates semantic ids: compact token sequences that are easier to generate and map back to catalogue items..

Semantic ids: The Middle Ground

A **Semantic id (Sid)** is a short sequence of tokens that identifies one catalogue item:

$$\text{item} \longleftrightarrow (z_1, z_2, \ldots, z_L).$$

Sid tokens are **shared across items**: related items can share some tokens, while the full sequence still identifies one item.

$$\text{item A} \longleftrightarrow (z_1{=}12,\ z_2{=}48,\ z_3{=}5,\ z_4{=}73)$$
$$\text{item B} \longleftrightarrow (z_1{=}12,\ z_2{=}48,\ z_3{=}19,\ z_4{=}91)$$

A few tokens that jointly index one item.

t3, t321, t643, t1011

Source: Hou et al., CIKM 2025 tutorial.

Prefix granularity. In hierarchical Sids, shared prefixes describe coarser item groups and later tokens refine toward a specific item:

$$(12, 48, *, *) \;\rightarrow\; (12, 48, 5, 73).$$

Items A and B share $(12, 48, \ldots)$, coarse similarity, but diverge later. The full tuple identifies the item, usually after **collision handling**.

Middle ground: shorter than a full description, more structured than one arbitrary token.

Semantic ids: Small Codebooks, Large Item Space

Core idea: use L token positions, each with K choices.

L = id length $\qquad$ K = codebook size per position

$$\underbrace{256 \times 256 \times 256 \times 256}_{L=4,\ K=256} = K^L = 256^4 \approx 4.3 \times 10^9$$

Only 4×256 code tokens can define billions of possible id sequences.

Why this helps

- Avoids one token per item.
- Reuses tokens across many items.
- Keeps generated ids short.

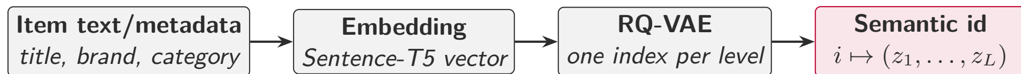
Important caveat

- Not every sequence is a real item.
- Valid ids come from the catalogue lookup table.
- Decoding can be constrained to valid ids.

Takeaway: semantic ids separate **capacity** from **vocabulary size**.

TIGER: Constructing the Semantic id (Offline)

TIGER (Rajput et al., NeurIPS 2023) builds an item-to-Sid lookup *before* training the generator; the tokenizer is then frozen.



Hierarchical codewords

$$\mathrm{SID}(i) = (z_1{=}7,\ z_2{=}1,\ z_3{=}4)$$

Earlier indices are coarser, later ones refine the residual: z_1 = broad category, z_2/z_3 = finer detail.

Collision handling Different items can map to the same tuple, so TIGER appends an extra token:

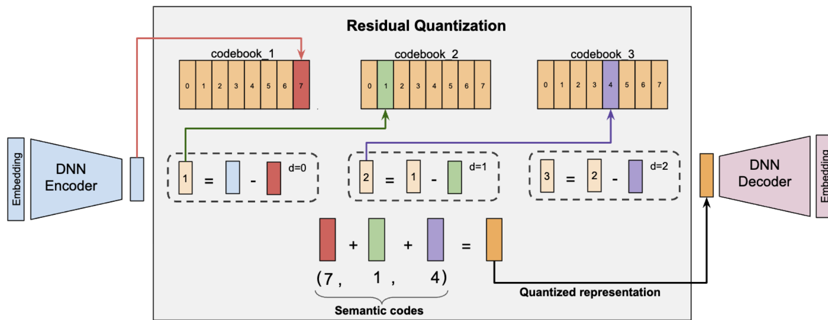
$$(12, 24, 52) \to (12, 24, 52, 0),\ (12, 24, 52, 1).$$

Each final Sid is unique and maps back to one catalogue item.

Key point: semantic-id construction is often an **offline tokenization step**. The recommender later generates the selected indices, not the continuous embeddings.

TIGER: RQ-VAE-Based Semantic IDs

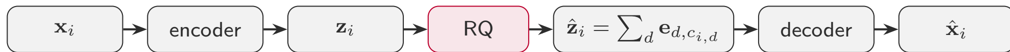
RQ-VAE: The **DNN encoder** maps an item embedding to a latent vector. Residual quantization then runs over L codebooks: at each level it picks the nearest codeword (the highlighted cell), *subtracts* it, and passes the **residual** to the next codebook. The chosen indices form the **Semantic id** $(7, 1, 4)$; their codeword vectors sum to the quantized representation, which the decoder reconstructs back to the embedding.



Source: Rajput et al., NeurIPS 2023. Toy illustration: $L = 3$ codebooks, $K = 8$ codes each.

TIGER: Training the RQ-VAE Tokenizer

Goal: learn an encoder, decoder, and residual codebooks so that discrete codeword indices preserve the original item embedding.



Semantic id: keep the selected codeword indices

$$\mathrm{id}(i) = (c_{i,1}, c_{i,2}, \ldots, c_{i,L})$$

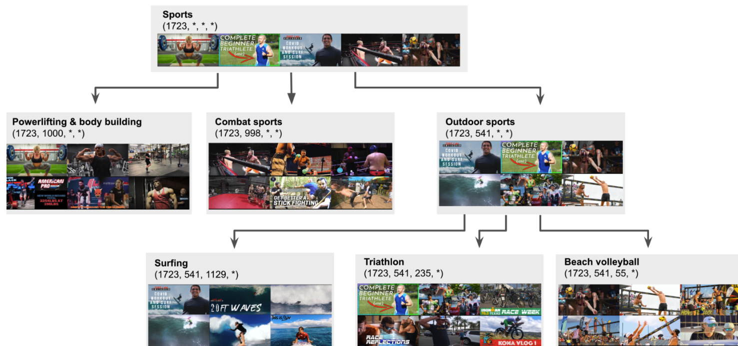
Training objective: the decoder reconstructs the original item embedding from the quantized latent vector:

$$\mathcal{L} = \mathcal{L}_{\text{recon}} + \mathcal{L}_{\text{rqvae}}, \qquad \mathcal{L}_{\text{recon}} = \|\mathbf{x}_i - \hat{\mathbf{x}}_i\|_2^2 .$$

After training, the decoder is not the recommender output: the generator predicts the **indices**, not the continuous vectors.

Semantic ids Form a Category Hierarchy

Shared prefixes = shared semantics. Items under the same coarse code sit in the same broad category; deeper codes split it into finer ones.



Source: Singh et al., RecSys 2024. $*$ = wildcard over deeper codeword positions.

Why it matters: the model can generate a coarse prefix (**Sports**) and refine token-by-token, this is what enables **cold-start** and **controllable, diverse** retrieval/recommendation.

Does the Item Identifier Choice Matter?

TIGER compares three item-id choices:

- **Random ids**
  arbitrary identifiers
- **LSH-based ids**
  hash similar embeddings to similar codes; random projections
- **RQ-VAE semantic ids**
  learned residual-quantized codes

Methods	Sports and Outdoors				Beauty				Toys and Games			
	Recall @5	NDCG @5	Recall @10	NDCG @10	Recall @5	NDCG @5	Recall @10	NDCG @10	Recall @5	NDCG @5	Recall @10	NDCG @10
P5 [8]	0.0061	0.0041	0.0095	0.0052	0.0163	0.0107	0.0254	0.0136	0.0070	0.0050	0.0121	0.0066
Caser [33]	0.0116	0.0072	0.0194	0.0097	0.0205	0.0131	0.0347	0.0176	0.0166	0.0107	0.0270	0.0141
HGN [25]	0.0189	0.0120	0.0313	0.0159	0.0325	0.0206	0.0512	0.0266	0.0321	0.0221	0.0497	0.0277
GRU4Rec [11]	0.0129	0.0086	0.0204	0.0110	0.0164	0.0099	0.0283	0.0137	0.0097	0.0059	0.0176	0.0084
BERT4Rec [32]	0.0115	0.0075	0.0191	0.0099	0.0203	0.0124	0.0347	0.0170	0.0116	0.0071	0.0203	0.0099
FDSA [42]	0.0182	0.0122	0.0288	0.0156	0.0267	0.0163	0.0407	0.0208	0.0228	0.0140	0.0381	0.0189
SASRec [17]	0.0233	0.0154	0.0350	0.0192	0.0387	0.0249	0.0605	0.0318	0.0463	0.0306	0.0675	0.0374
S^3-Rec [44]	0.0251	0.0161	0.0385	0.0204	0.0387	0.0244	0.0647	0.0327	0.0443	0.0294	0.0700	0.0376
TIGER [Ours]	**0.0264**	**0.0181**	**0.0400**	**0.0225**	**0.0454**	**0.0321**	**0.0648**	**0.0384**	**0.0521**	**0.0371**	**0.0712**	**0.0432**
	+5.22%	+12.55%	+3.90%	+10.29%	+17.31%	+29.04%	+0.15%	+17.43%	+12.53%	+21.24%	+1.71%	+14.97%

Source: Rajput et al., *Recommender Systems with Generative Retrieval*, NeurIPS 2023, Table 2.

Result in TIGER

In TIGER's experiments, **RQ-VAE semantic ids perform best** among these three identifier choices.

Caveat: later work shows that the best semantic-id construction method can depend on the **embedding space**, **task**, and **experimental setup**.

Lesson: item tokenization is not just preprocessing; it is a **modelling choice**.

Beyond TIGER: Semantic-id Design Is Still Open

Controlled tokenizer ablations

- TIGER establishes RQ-VAE as a strong starting point.
- Later work compares tokenizers under more controlled settings:
  - **RK-Means**
  - **R-VQ**
  - **RQ-VAE**
- Simpler residual-quantization methods can be competitive with, or better than, RQ-VAE.

Embedding space and task matter

- The embedding space used to build semantic ids affects performance.
- A tokenizer that works well for recommendation may not be best for search.
- In joint search–recommendation settings, task-specific ids can help one task while hurting the other.

Takeaway: RQ-VAE is a canonical starting point, but not a universal answer. **Semantic-ID construction is a modelling choice**: tokenizer, embedding space, and downstream objective all affect performance.

Background: Ju et al., *Generative Recommendation with Semantic IDs: A Practitioner's Handbook*, CIKM 2025; Penha et al., *Semantic IDs for Joint Generative Search and Recommendation*, RecSys 2025.

Semantic-id Construction: Main Design Families

Residual Quantization
ordered codes; coarse → fine
RQ-VAE, RQ-KMeans, RK-Means, R-VQ

Product Quantization
split embedding;
quantize subspaces
VQ-Rec

Hierarchical Clustering
tree-path ids; root → leaf
P5-CID, RecForest

LM / Textual ids
language tokens
or generated ids
LMIndexer, IDGenRec

Each family makes different trade-offs:

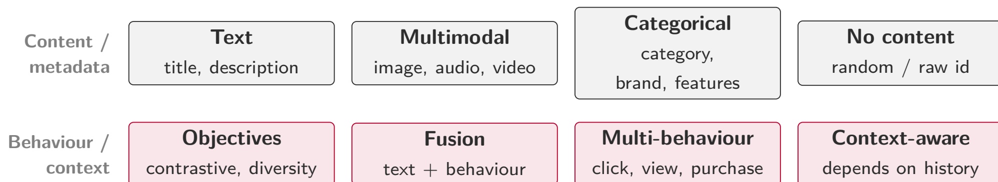
semantic content vs. **behaviour alignment** vs. **validity and decoding efficiency**

Where this leaves us

There is no universally best semantic-id construction method. The right choice depends on the **embedding space**, the **catalogue**, and the **downstream task**.

What Shapes a Semantic id?

Semantic ids are shaped by two things: **what representation we quantize** and **what objective we use to learn it**.



A Semantic id is only as good as the representation it quantizes.

Takeaway: the quantizer is only one part of semantic-id design. The field is moving from **static, content-only ids** toward **behaviour-aware, context-aware, and task-aware ids**.

Content Signals vs. Collaborative Signals

Semantic ids can be shaped by different sources of information. The key distinction is whether the signal comes from the **item itself** or from **user–item behaviour**.

Content signal

Information from item content:

title, description, category, brand, image

- Two movies have similar descriptions.
- Two products share a category or brand.

Collaborative signal

Information from interaction patterns:

clicks, views, purchases, ratings, co-consumption

- Users who watch *Inception* also watch *Interstellar*.
- Users who buy running shoes also buy running socks.

Why this matters: **content-based** ids capture what items *are*; **collaborative signals** capture how users *use items together*. **Behaviour-aware** tokenizers try to make item codes reflect both.

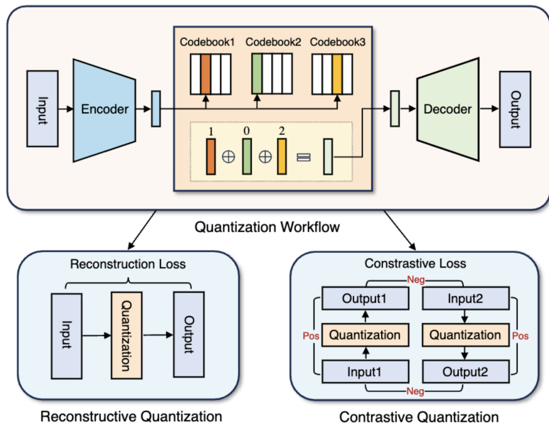
Beyond Reconstruction: CoST

Problem

Reconstruction-only quantization asks whether each item embedding can be recovered after discretization.

But recommendation also needs the tokenizer to preserve **neighbourhood structure**: which items should remain close after tokenization.

What CoST adds: a **contrastive objective** so the quantized representation stays closer to its own item embedding than to other items in the batch.



Source: Zhu et al., *CoST: Contrastive Quantization based Semantic Tokenization for Generative Recommendation*, RecSys 2024.

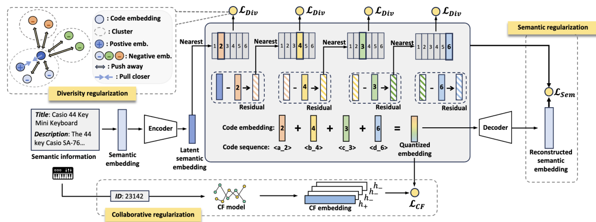
Behaviour-Aware Tokenization: LETTER

Problem

Content-based item ids can preserve textual or semantic similarity, but they may miss **collaborative signals**.

Items that users treat similarly are not always the items with the most similar text.

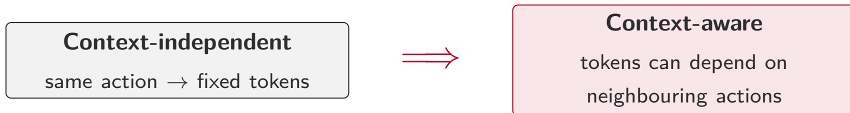
What LETTER adds: three regularizers: **semantic** hierarchy, **collaborative** behaviour alignment, and **diversity** for balanced code usage.



Source: Wang et al., *Learnable Item Tokenization for Generative Recommendation*, CIKM 2024.

Frontier: Context-Aware Tokenization (ActionPiece)

Problem. Most item-tokenization methods assign a fixed token sequence to each item or action. But the same action can carry different meaning depending on the surrounding sequence context.



Source: Hou et al., *ActionPiece: Contextually Tokenizing Action Sequences for Generative Recommendation*, ICML 2025.

ActionPiece. Represents each action as an unordered **feature set** and learns a **subword-style** vocabulary by merging feature patterns within actions and across adjacent actions. The same action can receive different tokenizations in different sequence contexts.

Coffee Break

15 min coffee break

We resume with training and decoding for generative recommendation.

Training and Decoding

Architecture: Encoder–Decoder or Decoder-Only?

Both are Transformers; they differ only in **how the history is read** before the next item is written.

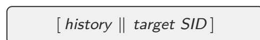
Encoder–decoder *"read fully, then write"*

e.g. T5-style; TIGER, LETTER, CoST

- *History encoded **once**, then decoded.*
- *Natural fit when history is **bounded**.*

Decoder-only *"one continuous stream"*

GPT-style; HSTU, OneRec, GPTRec



- *History and target are **one sequence**, predict the next token throughout.*
- ***Scales** to very long histories; industrial trend (2024–25).*

Either way, the item's code is produced one token at a time:

$$p(z_1, \ldots, z_L \mid h) = \underbrace{p(z_1 \mid h)}_{\text{first code}} \prod_{\ell=2}^{L} \underbrace{p(z_\ell \mid h,\, z_{<\ell})}_{\text{next code, given the ones so far}}$$

How Do We Represent Items?

Before training, we pick **what a token is**. Two dominant choices, driven by the catalogue.

Atomic IDs *one item = one token*

$$i_j \longrightarrow \langle \text{item}_j \rangle$$

- Vocab $= |\mathcal{I}|$; works for **finite, stable** catalogues.
- + simple, no tokenizer stage.
- – vocab explodes; **no cold-start generalization**; popularity bias in softmax.

e.g. GPTRec, P5, "Atomic IDs are enough" (arXiv:2508.10478).

Semantic IDs *one item = L codebook tokens*

$$i_j \longrightarrow (z_{j,1}, \ldots, z_{j,L})$$

- Small shared codebook ($K \sim 256$–4096); for **large, growing** catalogues.
- + compact vocab; warm cold-start; content-aware.
- – extra tokenization (Sec. 3); decoding must stay **valid**.

e.g. TIGER, LETTER, CoST, OneRec.

For the rest of this section

We assume **SIDs** (the harder case). Atomic IDs are the special case $L = 1$ with codebook = catalogue.

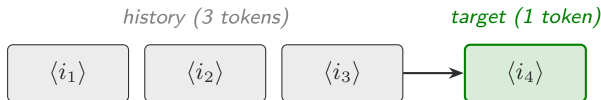
Building Training Examples — Atomic IDs

Input: chronological user interaction sequence.

$$(i_1, i_2, \ldots, i_t, \;\; i_{t+1}) \qquad \text{predict } i_{t+1} \text{ from } (i_1, \ldots, i_t).$$

With **atomic IDs**, each item is a single token in the vocabulary:

$$i_j \;\longrightarrow\; \langle i_j \rangle.$$



Learning problem:

$$p(i_{t+1} \mid i_1, \ldots, i_t).$$

Note

One item = one position. Predicting the next item is exactly **one** autoregressive step. Vocabulary = catalogue.

Building Training Examples — Semantic IDs

With **Semantic IDs**, each item expands into L codebook tokens:

$$i_j \longrightarrow (z_{j,1}, z_{j,2}, \ldots, z_{j,L}).$$

The history becomes an $L\times$ **longer** flat token sequence:

Learning problem:

$$p(z_{t+1,1}, \ldots, z_{t+1,L} \mid \text{history of SID tokens}).$$

Note

One item $= L$ positions. Predicting the next item now takes L autoregressive steps — this is what beam search will operate on.

Cold-Start at Inference: New Items, No Retraining

A new item $i^\star$ arrives *after* the model is trained. Two very different stories:

Atomic IDs

- Token $\langle i^\star \rangle$ does **not exist** in the vocabulary.
- Embedding row must be added and **learned** from interactions.
- Until then: $i^\star$ is unrecommendable (**strict cold start**).
- Fix: periodic retraining / embedding warm-up from content.

Semantic IDs

- Run $i^\star$ through the (frozen) **tokenizer** from Sec. 3 $\rightarrow$ get SID $(z_1^\star, \ldots, z_L^\star)$.
- All sub-tokens already exist in the codebook.
- Add the path to the **trie**; $i^\star$ is now **decodable**.
- Similar items share prefixes $\Rightarrow$ **generalization** for free.

Training Stages: Grounding the SID Vocabulary

The cold-token problem. When we extend an LM's vocabulary with $K \cdot L$ fresh SID tokens, those embeddings are **randomly initialized**. Jumping straight to next-item CE forces the model to learn (i) what each SID *means* and (ii) how to compose them, from a single signal.

Fix: a **grounding stage** before next-item training. Teach the model to use SIDs in many contexts so the embeddings absorb meaning.

Typical grounding tasks

(formulated as text-to-text, item $i \leftrightarrow$ SID(i)):

- **SID $\to$ description**
  "Item $(12, 48, 7)$ *is:"* $\to$ *title/desc.*
- **Description $\to$ SID**
  "A sci-fi film by Nolan" $\to$ $(12, 48, 7)$.
- **Attribute / category alignment**
  "Genre of $(12, 48, 7)$*?"* $\to$ *"Sci-fi".*
- **Co-occurrence / similarity**
  "Items similar to $(12, 48, 7)$*:"* $\to$ *list.*

Three-stage pipeline

1. **Tokenize**: learn SIDs (Sec. 3).
2. **Ground**: multi-task tuning so SID embeddings carry semantic + collaborative signal.
3. **Recommend**: next-item CE (Sec. 4) + optional RL (next frame).

Stages can be sequential or jointly multi-task.

Training Objective: Next-Token Cross-Entropy

The model is trained with standard **next-token prediction**:

$$\mathcal{L} = -\sum_{\ell=1}^{L} \log p(z_\ell \mid \text{history},\ z_{<\ell})$$

What this means concretely:

- At training time, the target SID is known.
- For each position ℓ, predict z_ℓ given everything before it (**teacher forcing**).
- Loss is averaged over all positions and all items in the batch.

What's the same as a language model?
Everything about the loss function and training loop.

What's different?
The tokens are item codes drawn from a **small, learned codebook** (∼256–4096 entries), not a 50K BPE vocabulary.

Important distinction

The model is **not** generating natural language. It is generating **item identifiers**.

Beyond Cross-Entropy: Reward-Based Fine-Tuning

The problem with next-token CE: it only rewards copying the *exact* item the user clicked next. It never tells the model whether a recommendation is a **real item**, whether the *whole list* is good, or whether it meets goals like diversity and freshness.

The idea: let the model *generate* a few recommendations, *score* them, and then *nudge* it to produce more of the good ones, the same way chatbots are tuned with human feedback. This is **reinforcement learning (RL)**.

The recipe (GRPO). For one user history, repeat:

1. **Generate** a small group of candidate recommendations.
2. **Score** each one with a *reward* (higher = better).
3. **Compare** each candidate to the group's *average* score:
   above average ⇒ make it **more** likely; below ⇒ make it **less** likely.
4. Take a **small, careful** step in that direction (more on the next slide).

Why "compare to the group"?

We don't need to know the "true" best recommendation, only which candidates are better *than the others we just tried*. That relative signal is enough to improve, and it needs no extra scoring network.

Reward-Based Fine-Tuning: A Concrete Example

What goes into the reward? A recommendation scores higher when it is **valid** (a real catalogue item), **relevant** (matches what the user wants), and meets goals like **freshness, diversity, and safety**.

Example: a user who watched *Inception, Interstellar, Tenet*

The model generates 4 candidates and we score each from 0 (bad) to 1 (great):

Oppenheimer	score 1.0	valid, relevant, fresh
Dunkirk	score 0.7	valid, relevant
Interstellar	score 0.2	already watched
made-up code	score 0.0	not a real item

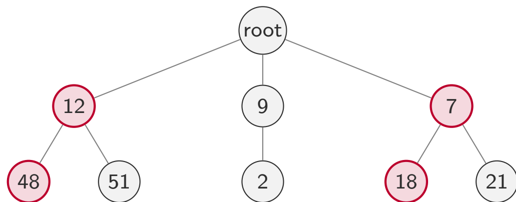
Group average $= 0.475$. So *Oppenheimer* and *Dunkirk* are **above average** (push up); *Interstellar* and the invalid code are **below average** (push down).

Two safety rails when we "nudge": take only **small steps** each update, and **stay close to the original** model, so it can't drift into nonsense just to chase reward.

Inference: Beam Search Over SIDs

Greedy decoding keeps only the top-1 next token at each step, fine for the next item, but we want a *ranked list*.

Beam search maintains B partial candidates at each step:



Beam size $B = 3$. *Red nodes are kept at each level; gray are pruned.*

After L steps, B complete SIDs come out:

$$(12, 48, 5), \quad (12, 48, 7), \quad (7, 18, 3) \quad \longrightarrow \quad B \text{ ranked items}$$

The Validity Problem

Language generation: any token sequence is a valid (if perhaps weird) sentence.

Recommendation generation: most SIDs **don't correspond to any real item**.

Why this happens:

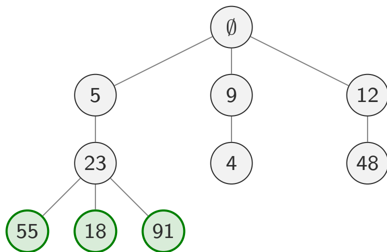
- The SID space has K^L possible codes (e.g., $\sim 10^9$).
- The catalogue uses only a tiny fraction (e.g., $\sim 10^7$ items).
- Most code combinations are **unused**.

The fix

Constrain decoding so the model can only emit token sequences that exist in the catalogue.

Trie-Constrained Decoding

Store all valid catalogue SIDs in a **trie**. At each decoding step, only allow tokens on a valid path.



At prefix $(5, 23)$:

- Allowed next tokens: $\{55, 18, 91\}$
- Forbidden: everything else.

Effect:

- Every generated sequence **is** a real item.
- The model's output distribution is renormalized over allowed tokens only.
- Implementation: a logit mask at each step.

Trade-off

Validity is guaranteed, but the trie must be *updated* every time the catalogue changes, might be non-trivial in fast-moving systems.

Another Angle: Reward Validity Instead of Masking It

The trie *forces* validity. A complementary idea: **teach** the model to prefer valid SIDs on its own, by making "is this a real item?" part of the GRPO **reward**, candidates are generated *freely* (no mask), valid ones rewarded, invalid ones penalized.

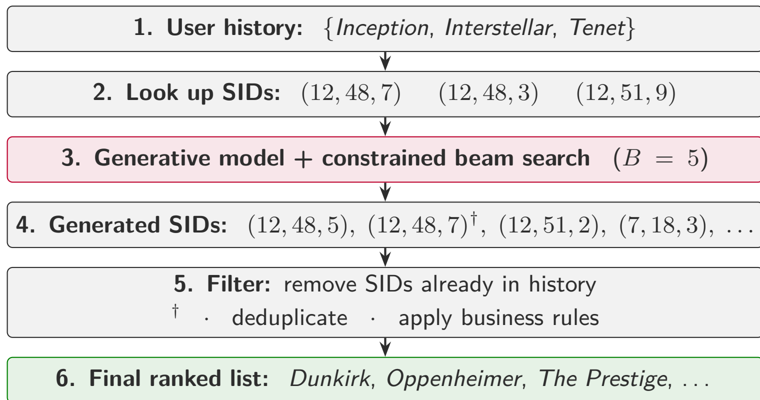
Example (validity-only reward: 1 = real item, 0 = not)

$(12, 48, 7) = 1.0$ real item
$(12, 48, 5) = 1.0$ real item
$(5, 99, 13) = 0.0$ no such item
$(12, 51, 2) = 1.0$ real item

$(5, 99, 13)$ is **below average** $\Rightarrow$ generated **less**; valid ones pushed **up**.

Trie vs. reward: trie = **guaranteed** but needs syncing; reward = only *likely* valid but no live trie needed. Often **combined**.

End-to-End: From History to Ranked List



† *The generated SID matches Interstellar, already in history; filter it out.*

*With **atomic IDs** the flow is identical, but step 2 maps each item to a single token and step 4 generates one token per recommendation.*

Decoding Is Not Free

LLM-style beam search has **known pathologies** that hit GenRec especially hard:

Amplification bias

Popular SID prefixes (e.g. $(12, 48, \cdot)$) dominate beam search; long-tail items get pruned early.

Local optima

Greedy first-token choice locks in a region of SID space; a better item with a different prefix is unreachable.

Homogeneity

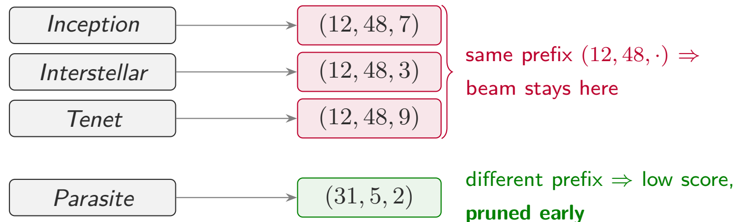
Beam candidates share long prefixes, so the top-B items are nearly identical (e.g. five similar action movies).

Inference cost

Each recommendation $= L$ autoregressive steps $+$ trie lookup. At scale, decoding dominates latency.

The Problem: All Recommendations Look the Same

Root cause: similar items get **similar SIDs** : they share the same opening codes. Beam search locks onto that popular prefix and never leaves, so the whole list collapses into one “family”.



Goal: keep recommendations **valid and relevant**, but spread them across **different prefixes**, so the user actually has a choice.

Two Ways to Add Diversity

At decoding time (change *how we pick*):

- **Add randomness** (temperature / sampling): don't always take the single most likely item.
  e.g. sometimes pick the 3rd-best opening, so a comedy can sneak into a list of thrillers.
- **Force groups to differ** (diverse beam search): split the candidate lists into groups and penalize a group for repeating earlier choices.
- **Re-rank afterwards** (MMR): build the list one item at a time, each time preferring an item *unlike* those already chosen.
  e.g. after picking Inception, skip Tenet in favour of something new.

At training time (change *what we learn*):

- **Reward diversity in RL**: in the GRPO group, penalize candidates that look too much like each other.
  e.g. five near-identical action films share the reward; spreading out scores higher.
- **Fix it at the tokenizer** (LETTER): design SIDs so popular items don't all collapse onto the *same prefix* in the first place.

The trade-off

More diversity usually means slightly **lower accuracy** at guessing the exact next click, but higher **novelty** and real-world engagement. The right balance depends on the product, not the benchmark.

Design Choices, at a Glance

Choice	Options (with examples)
Item representation	**Atomic IDs** — GPTRec, P5 | **Semantic IDs** — TIGER, LETTER, OneRec
History length	Short (∼50) — TIGER | Long (∼1000+) — HSTU, OneRec
Architecture	Encoder–decoder — TIGER, OneRec | Decoder-only — HSTU, OneRec-V2
Training stages	CE only — TIGER | **Grounding** + CE — LC-Rec, LETTER | CE + **RL/DPO** — OneRec, S-DPO, Rec-R1
Decoding	Greedy | Beam | **Constrained beam** | Sampling
Post-processing	Filter | Dedup | Business rules | Re-rank

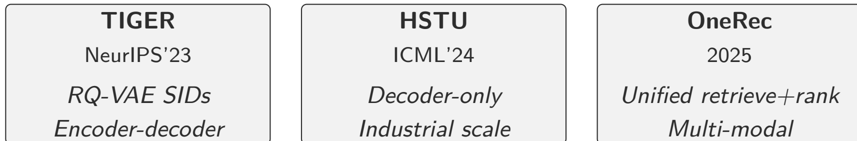
Recap

Training pipeline: tokenize → **ground** SID embeddings → next-token CE → optional **RL/DPO** for validity, listwise quality, business goals.
Inference: **constrained** beam search to guarantee valid items.
Decoding pathologies (bias, homogeneity, cost) shape modern GenRec research.

Limitations, Open Challenges, and Outlook

Where Are We Now?

Generative recommendation works. Three frameworks established the paradigm:



But moving from *retrieve-and-rank* to *generate* **changes the problem**. Some things become **harder**; others become **newly possible**.

This section

We close the lecture with two complementary lenses on the road ahead:

- **What becomes harder** : the limitations that GenRec inherits or creates.
- **What becomes possible** : the new capabilities GenRec unlocks.

What Becomes Harder

Limitations of Generative Recommendation

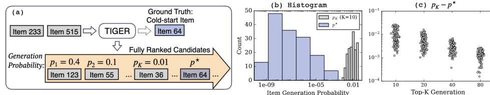
Harder #1: Cold-Start Becomes Fragile

The promise: a brand-new item gets a valid SID from its *content* the moment we tokenize it, so in principle it can be recommended right away, no clicks needed.

The catch: being **decodable** (a valid SID exists) is not the same as being **recommended**. The model was **trained only on SIDs of items people actually clicked**, so it learned to output those. A fresh item's SID has almost **no probability**, and beam search prunes it before it is ever considered.

Fix: stop relying on the generator to "think of" cold items, hand them to a dense model instead.

1. **Generate** the usual candidates (will be mostly warm items).
2. **Inject** cold items by hand into the candidate pool.
3. **Re-rank** everything with **dense embeddings**, which compare items by content, so cold items get a fair score.



Source: Yang et al., *LIGER*, 2024.

Lesson

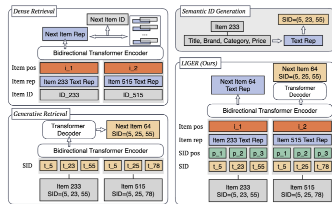
The generator alone won't surface cold items; pairing it with a dense re-ranker does. The future of GenRec may be **hybrid**, not purely generative.

Harder #2: Dense Retrieval Is Still Strong

The "old" approach hasn't gone away. **Dense retrieval** stores one embedding per item and finds neighbours of the user's vector; generative retrieval instead *generates* item codes. On many academic benchmarks, dense is still as good or better.

Dense retrieval

+ Strong ranking, simple to train and serve.
+ Cold-start is easy: a new item just gets a **text embedding**.
– Must store *every* item vector and search them (approximate nearest-neighbour) : costly at billions of items.



Source: Yang et al., 2024.

Generative retrieval

+ Compact SIDs; generates candidates without scanning the whole catalogue.
– **Cold-start is fragile** (Harder #1).
– Ranking quality is less consistent.

Takeaway: generative retrieval is **not strictly better**, it trades storage/search cost for cheaper generation, at some cost to ranking and cold-start.

Harder #3: Decoding Is Expensive and Biased

Generating recommendations one code at a time (Sec. 4) creates two distinct headaches:

Biased *(what it shows)*

- **Popularity amplification:** popular code prefixes win the beam, so niche / long-tail items are **pruned early** and rarely surface.
- **Homogeneity:** the top results share a prefix, so the list is full of **near-duplicates**.

Expensive *(what it costs)*

- **Latency:** each recommendation needs L sequential steps, hard to answer within the $<$**50 ms** budget real systems require.
- **Trie upkeep:** the validity trie must stay in sync with a catalogue that changes constantly.

How research is attacking the cost:

- **Speculative decoding**: a small, cheap model drafts the codes; the big model only *verifies*, fewer expensive steps.
- **Parallel generation** (RPG): emit all L codes *at once* instead of one after another.
- **Caching**: popular prefixes are recomputed millions of times, so cache and reuse them.

(Bias / diversity fixes were covered in Sec. 4.)

Harder #4: Catalogues Move; Evaluation Lags

Two practical problems that don't show up on a static research benchmark but dominate real deployments.

1. The catalogue keeps changing

Unlike a fixed dataset, real catalogues add and drop items constantly (TikTok, news, shops). This stresses the SID machinery:

- **New items:** they need a SID, but the tokenizer (quantizer) was trained on the *old* catalogue. Re-running it for every new item is fine; **retraining** it would shift *all* existing SIDs.
- **Removed items:** their SIDs linger in the model's "vocabulary" : does it stop recommending them, or keep suggesting dead products?

2. Our metrics don't fit generation

Offline metrics (Recall@K, NDCG@K) ask: *did we predict the one item the user actually clicked?*

- A generative model may surface a **good item the user never saw** : it counts as **wrong** simply because it isn't the logged click.
- So benchmarks can **under-credit** exactly the novelty we built GenRec for.
- What we actually care about, **diversity, novelty, fairness, long-term engagement**, isn't captured at all.

Harder #5: Safety, Privacy, and Governance

Generative recommenders inherit **all the safety problems of LLMs**, on top of the classical RecSys ones.

Content & policy

- Decoder can emit a *valid SID* for an item that is **NSFW, deprecated, region-locked, or recalled**.
- The **trie** is the only hard safety net : it must be filtered *per request* (per user, per locale).
- No equivalent to a re-ranker's blocklist if you skip the trie.

Privacy & memorization

- SIDs are derived from item content ⇒ **content leakage** risk in the codebook.
- Long user histories used as context can be **memorized** by the decoder (cf. LLM extraction attacks).
- GDPR *right to be forgotten*: how do you unlearn an item from a frozen tokenizer?

Auditability

Simple non LLM-based ranker can explain *why* item i was retrieved (which features fired). A decoder emitting $(12, 48, 7)$ offers no such trace : **explanation** becomes a generation task of its own.

The honest summary

We are deploying **LLM-shaped systems** into a domain (recommendation) with **LLM-shaped risks**.

What Becomes Possible

New Capabilities Unlocked by Generation

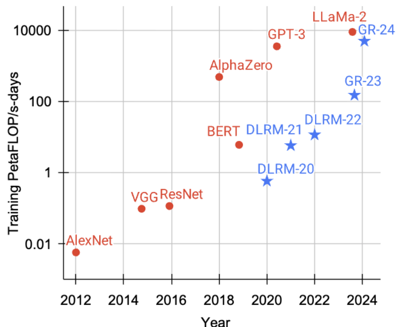
Possible #1: Scaling Laws for Recommendation

What is a "scaling law"? For LLMs, more **data + compute + parameters** reliably means a better model, no plateau. Classic recommenders **don't** behave this way: past a point, making them bigger barely helps.

The question: can we make recommenders that **keep improving as we scale**, like LLMs do?

Actions Speak Louder than Words (Zhai et al., ICML'24) says **yes**, by treating the stream of **user actions** (clicks, watches) like the stream of *tokens* an LLM is trained on, and using a Transformer (**HSTU**) built for very long histories.

Result: performance keeps climbing with compute (right), and it **beats the heavily-tuned production model** (a "DLRM").



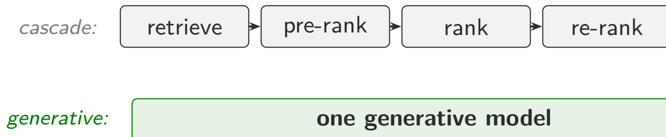
Source: Zhai et al., ICML 2024.

The shift

From small *task-specific* recommenders that plateau → **large generative recommenders** that get better the more we scale them, just like language models.

Possible #2: One Model, Many Tasks

The cascade splits a problem the generative view can **unify**:



And the same backbone can do more:

- Sequential recommendation, search, query suggestion, explanation generation.
- Multi-domain transfer (books → movies, etc.) via shared SID vocabulary.
- Combined with pretrained LLMs (LC-Rec, OneRec-Think): zero-shot generalisation, instruction following.

The reframe

RecSys becomes **a sequence modelling task**, like NLP. The whole LLM toolkit becomes available.

Possible #3: Instruction-Based Recommendation

Once recommendation is sequence generation, the user can speak to the model in **natural language** ˇ2014 not just by clicking.

The shift

- Classical: input is a *history* $(i_1, \ldots, i_t)$.
- GenRec + LLM: input is a *history* + *instruction* *"something upbeat for a morning run, no true-crime"*.
- Output is still a constrained SID sequence $\Rightarrow$ catalogue-grounded, no hallucination.

Why it matters

- Captures **intent**, not just long-term preference.
- Enables **controllability**: explicit diversity, mood, novelty knobs.
- Unifies search and recommendation under one interface.

Case study: GLIDE (Spotify, 2026)

- Podcast discovery as **instruction-following** over SIDs.
- Recent listening + lightweight context as prompt.
- Long-term user embedding injected as a **soft prompt** *(stable preference under tight latency budget)*.
- Trie keeps generation grounded in the catalogue.

Instruction-Based Recommendation: Example Prompts

The same model and catalogue, driven by very different natural-language requests. Each instruction is added to the user's history; the output is still a **constrained SID sequence**, so it stays grounded in real catalogue items.

Example instructions a user could type

- *"A 20-minute true-crime podcast for my commute."* (length + topic + occasion)
- *"More like the last one, but lighter and funnier."* (refine the previous result)
- *"Cozy movies for a rainy Sunday, nothing scary."* (mood + a hard constraint)
- *"Surprise me with something outside my usual taste."* (explicit novelty knob)
- *"Albums similar to this one but in Spanish."* (similarity + attribute filter)

The point: none of these are clicks. The user states **intent** directly, and the model turns it into grounded recommendations, something a classical history-only recommender cannot do.

Possible #4: Test-Time Reasoning for RecSys

LLMs got much better by **thinking step-by-step** before answering (chain-of-thought, o1-style models), spending extra compute *at prediction time*. **Can a recommender "think" before suggesting the next item?**

Think Before Recommend (Tang et al., 2025): instead of jumping straight from history to a prediction, the model takes **several internal refinement steps** first.

$$\text{user history} \rightarrow \text{think}_1 \rightarrow \text{think}_2 \rightarrow \cdots \rightarrow \text{next item}$$

(The "thinking" happens inside the model's hidden states, it is not written-out text.)

Why thinking could help (intuition)

History: cooking videos, then a flight-booking app, then a Rome guide. A one-shot model sees mixed signals; a few reasoning steps can infer *"planning a trip to Italy"* and recommend accordingly.

Reported gains (NDCG@20, across several standard recommenders):

SASRec: **+9%** · BERT4Rec: **+6%** · UniSRec: **+7%** · MoRec: **+3%**

If reasoning were perfect (an "oracle"), gains reach +37% to +53%, so lots of headroom remains.

Source: Tang et al., *Think Before Recommend*, arXiv 2503.22675.

Key Takeaways

How a GenRec model works (Sec. 4)

1. It reframes recommendation as **sequence generation**: from scoring $s(u, i)$ to generating the next item's code $p(z_1, \ldots, z_L \mid \text{history})$.
2. **How items are represented** is the key choice: atomic IDs (one token each) vs. **Semantic IDs** (a few shared codebook tokens, compact and content-aware).
3. **Training:** ground the SID embeddings, then next-token cross-entropy, optionally RL (GRPO) to reward validity, diversity, and business goals.
4. **Decoding is part of the model**, not plumbing: trie-constrained beam search guarantees *valid* items; diversity tricks fight look-alike lists.

Where it stands & heads (Sec. 5)

5. **Harder than it looks:** cold-start can be **fragile** and dense retrieval is still strong on academic benchmarks, so the best systems are often **hybrid** (LIGER).
6. **New operational costs:** expensive/biased decoding, ever-changing catalogues, metrics that under-credit novelty, and inherited **LLM-shaped** safety/privacy risks.
7. **What it unlocks:** scaling laws for recommenders, *one* model for retrieve+rank+search, instruction-based recommendation, and test-time reasoning.

One line

Generative recommendation makes RecSys a **sequence-modelling** problem, unlocking the LLM toolkit, but the trie, the tokenizer, and decoding become first-class parts of the system.