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

Motivation: From Ranking to Generation

Generative Recommendation

Semantic IDs and Tokenization

Training and Decoding

Limitations, Open Challenges, and Outlook

Recommendation as Sequence Generation

$$\underbrace{i_1, i_2, \ldots, i_t}_{\text{user interaction history}} \longrightarrow \underbrace{\mathrm{SID}(i_{t+1}) = (z_{t+1,1}, z_{t+1,2}, \ldots, z_{t+1,L})}_{\text{identifier of } i_{t+1}}$$

Notation. $i_1, \ldots, i_t$ are past interacted items; i_{t+1} is the next item to predict; $\mathrm{SID}(i_{t+1})$ is its Semantic ID; L is the SID length; $z_{t+1,\ell}$ is the ℓ-th token of that SID.

Core idea

- A generative recommender predicts $\mathrm{SID}(i_{t+1})$: a short token sequence that identifies the next item, conditioned on the user's past interactions.
- These interactions may include items the user **clicked, watched, purchased, listened to, or otherwise engaged with**.

The generated token sequence is later mapped back to a real catalogue item.

Probabilistic View: Generate an Identifier

$$\mathcal{H}_t = (i_1, i_2, \ldots, i_t), \qquad \mathbf{z}_{t+1} = \mathrm{SID}(i_{t+1}) = (z_{t+1,1}, z_{t+1,2}, \ldots, z_{t+1,L})$$

Generative recommendation

Instead of directly scoring catalogue items, the model predicts the next item's identifier autoregressively:

$$p(\mathbf{z}_{t+1} \mid \mathcal{H}_t) = \prod_{\ell=1}^{L} p(z_{t+1,\ell} \mid \mathcal{H}_t, z_{t+1,<\ell}).$$

After generation

$$(z_{t+1,1}, z_{t+1,2}, \ldots, z_{t+1,L}) \xrightarrow{\text{id-to-item lookup}} i_{t+1}$$

Motivation: From Ranking to Generation

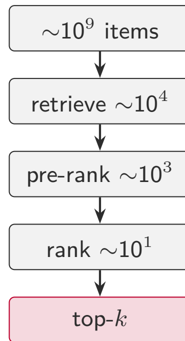
Recap: Classical Recommendation Pipelines

The task: sequential recommendation

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

Classical solution: score, then rank

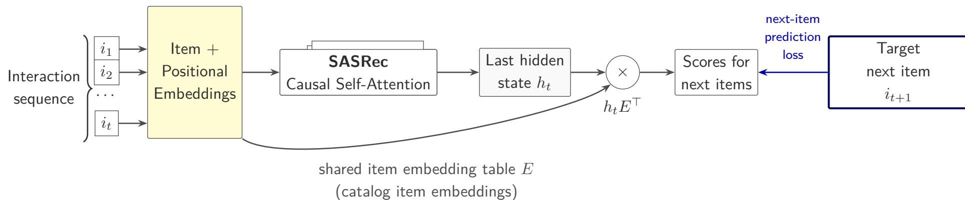
- **Score** candidate items given the history: $s(\mathcal{H}_t, i)$.
- **Rank** by predicted relevance: $\hat{i}_1, \hat{i}_2, \ldots, \hat{i}_k$.
- At scale, use a **cascade**: retrieve $\rightarrow$ pre-rank $\rightarrow$ rank.



Illustrative cascade system.

This pipeline works well and remains common in large-scale systems. However, retrieval, pre-ranking, and ranking often use different models, features, and objectives. Early-stage errors can therefore limit what later stages are able to recover.

Classical Sequential Recommendation: Score Every Item



- SASRec encodes the ordered interaction history with causal self-attention.
- The final hidden state h_t is matched against the item embedding matrix E:

$$s = h_t E^\top .$$

- This produces scores over catalogue items; training increases the score of the true next item i_{t+1}.

Key point

Classical sequential recommendation learns a history representation and uses it to **score items directly**.
Next: what if the model generated the item identifier instead?

Why Reframe Recommendation as Generation?

- Generative models are strong **sequence models**; user behaviour is also sequential.
- This motivates a natural question:

Can we treat recommendation as a generation problem?

Potential benefits

- Learn directly from large-scale behavioural sequences.
- Combine item text, images, audio, and user actions in one modelling framework.
- Reduce reliance on many separately trained retrieval and ranking stages.

The hard part

- Items are harder to tokenize than words:

$$|\text{language vocabulary}| \ll |\text{item catalogue}|$$

- Item catalogues are huge, dynamic, and non-stationary.

Key point

The central challenge is **item tokenization**: how do we turn catalogue items into tokens that a model can generate, while still being able to map the generated token sequence back to a real item?

Core GenRec framing: Rajput et al., NeurIPS 2023; practical SID design: Ju et al., CIKM 2025.

From LM Ideas to Generative Recommendation

Lecture 3 recap: ideas from language modelling and Transformers have entered RecSys in several ways.

Architecture

Transformer over item sequences

SASRec, BERT4Rec

Representation

LM encodes item text or metadata

Recformer, item-text encoders

Task formulation

Recommendation framed as language tasks

P5, M6-Rec

Generative recommendation = generate item identifiers

user history $\longrightarrow$ next-item identifier

Examples: TIGER, OneRec

Positioning

Generative recommendation is the formulation where the model **generates the identifier of an item**, rather than only using language models for representation or scoring.

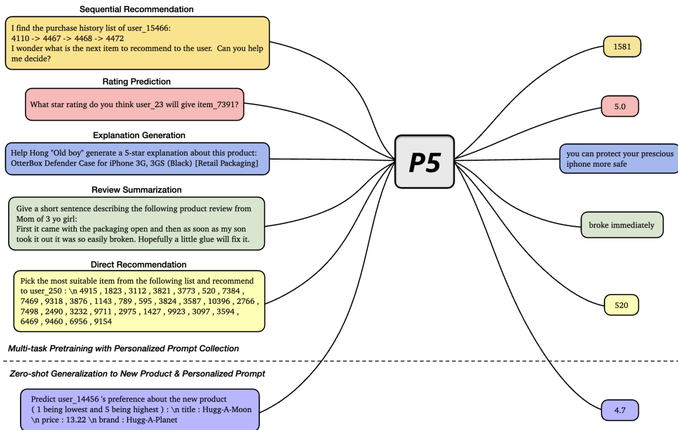
Background: SASRec (Kang & McAuley, 2018); BERT4Rec (Sun et al., 2019); RecFormer (Li et al., 2023); P5 (Geng et al., 2022); M6-Rec (Cui et al., 2022); TIGER (Rajput et al., 2023); OneRec (Deng et al., 2025).

Historical Bridge: P5 – Recommendation as NLP

- P5 frames multiple recommendation tasks as text-to-text language-processing tasks.
- Tasks are written as prompts:
  - sequential recommendation,
  - rating prediction,
  - explanation,
  - review summarization.
- This connects recommendation with sequence-to-sequence language modelling.

Bridge to GenRec

P5 shows that recommendation tasks can be cast as **text-to-text generation**; GenRec makes the output **catalogue-grounded item identifiers**.



Source: Geng et al., *Recommendation as Language Processing (RLP): A Unified Pretrain, Personalized Prompt & Predict Paradigm*, RecSys 2022.

What Should the Item Identifier Look Like?

- To recommend by generation, the model first needs a token representation of each item:

$$i \to (z_1, z_2, \ldots, z_L)$$

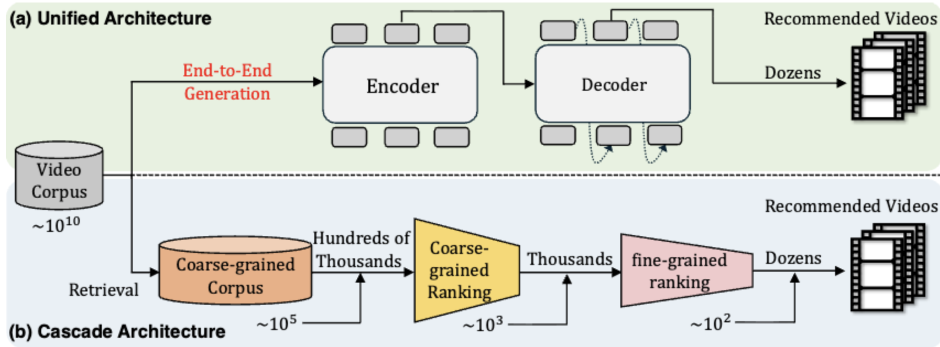
- This representation should be:
  - **compact** enough to generate efficiently,
  - **expressive** enough to capture item semantics and behaviour,
  - **learnable** enough for the model to predict reliably,
  - **valid** so generated sequences map back to catalogue items.
- This is the central tokenization problem in generative recommendation.

Next question

Can we design item identifiers that are compact, meaningful, learnable, and grounded in the catalogue?

Background: Rajput et al., NeurIPS 2023; Penha et al., RecSys 2025; Ju et al., CIKM 2025.

The Big Picture: From Cascades to Generated Identifiers



Source: Deng et al., *OneRec: Unifying Retrieve and Rank with Generative Recommender and Iterative Preference Alignment*, arXiv 2025.

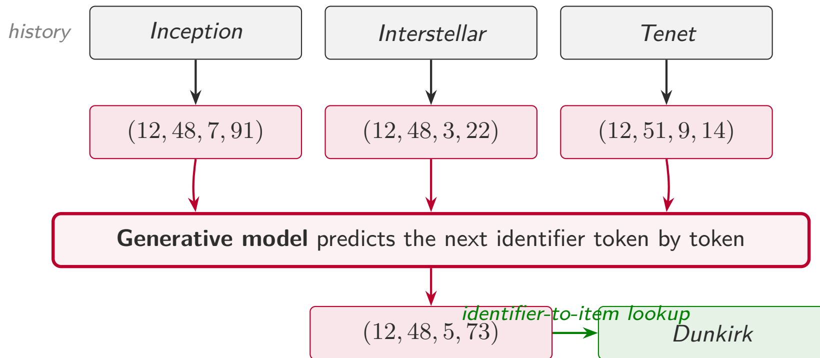
Three questions for generative recommendation

(1) How do we tokenize catalogue items?
(2) How do we train a model to generate item identifiers?
(3) How do we keep generated identifiers valid and grounded in the catalogue?

Generative Recommendation

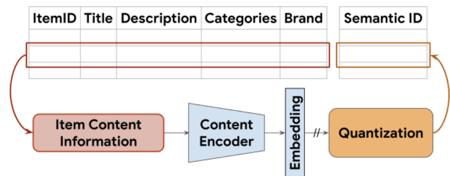
Walkthrough: One Recommendation, End to End

A toy example: a user has watched three movies, and the model generates the identifier of a candidate next recommendation. **movie history** $\longrightarrow$ **SID history** $\longrightarrow$ **generated SID** $\longrightarrow$ **catalogue item**

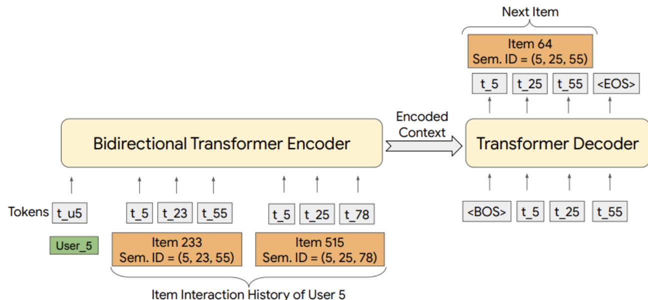


- Each catalogue item is represented by a short token sequence, called a **Semantic ID**.
- The model reads the Semantic IDs of past items and **generates** the Semantic ID of the next item.
- The generated Semantic ID is mapped back to a real catalogue item by lookup.

Two Stages: Assign Identifiers, Then Generate



(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.

Two Stages in SID-Based Generative Recommendation

Stage 1 – Item Tokenization

- Usually pre-computed for the current catalogue.
- Each item is mapped to a sequence of discrete tokens:

$$i \rightarrow (z_1, z_2, \ldots, z_L)$$

- The tokens can be learned from item text, metadata, behaviour, or multimodal content.

Stage 2 – Autoregressive Generation

- The user history is represented using item-token sequences.
- The model predicts the next item's identifier one token at a time:

$$p(z_\ell \mid \text{history}, z_{<\ell})$$

- Decoding plus identifier-to-item lookup returns candidate items.

Main point

GenRec first defines the item token space, then learns to generate item identifiers from user histories.

The Full Semantic-ID GenRec Pipeline

Offline: build the SID space

1. Collect item content or signals
   title, description, image, behaviour, . . .
2. Encode each item as a dense vector:

$$i \to \mathbf{e}_i$$

3. Tokenize the embedding into a Semantic ID:

$$\mathbf{e}_i \to (z_1, \ldots, z_L)$$

Training

4. Rewrite user histories as SID-token sequences.
5. Train a sequence model to generate the next SID:

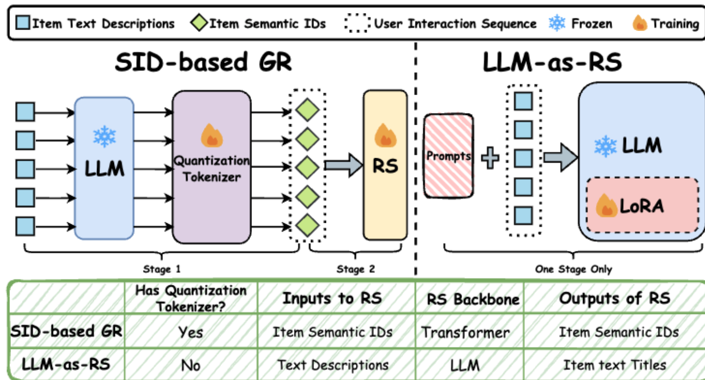
$$p(z_\ell \mid \text{history}, z_{<\ell})$$

Inference

6. Represent the user's recent history as SID tokens.
7. Generate candidate SIDs, usually with beam search.
8. Constrain or filter to valid catalogue SIDs.
9. Map SIDs back to items, optionally re-rank, and present top-k.

Step 3 is the hinge.
The quality of the Semantic ID space strongly shapes everything downstream: training, generation, and ranking.
Next section: how do we build it?

Two Paradigms for 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

Big picture. There are two main generative recommendation paradigms: **SID-based GenRec**, which first builds Semantic IDs and then generates them, and **LLM-as-RS**, which directly uses item text in a one-stage LLM-based setup.

Adapted from Figure 1 in *Understanding Generative Recommendation with Semantic IDs from a Model-Scaling View*, ICLR 2026 submission.

SID-based GenRec vs. LLM-as-RS

SID-based GenRec

- First tokenize items into Semantic IDs:

$$i \to (z_1, \ldots, z_L)$$

- Train a sequence model to generate the next SID:

  history SIDs → next SID

- **Pros:** compact outputs, efficient decoding, clear catalogue grounding.
- **Challenge:** the Semantic ID itself may become a bottleneck.
- Example: **TIGER**.

Our main focus: how item identifiers are constructed and generated.

LLM-as-RS / Decoder-only

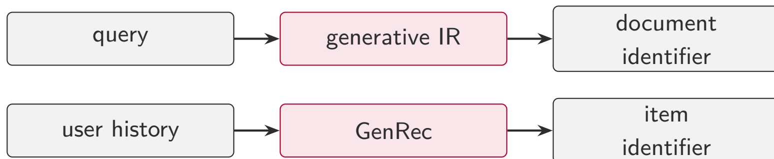
- Use item text or descriptions directly as input.
- Fine-tune an LLM-style recommender on user histories.
- Generate the next item's text, title, or identifier.
- **Pros:** richer direct use of language-model knowledge.
- **Challenge:** usually more expensive to train and serve.

Useful contrast: avoids a separate SID tokenizer, but sacrifices efficiency.

Background: Rajput et al., NeurIPS 2023; Ju et al., CIKM 2025; *Understanding Generative Recommendation with Semantic IDs from a Model-Scaling View*, ICLR 2026 submission.

A Useful Parallel: Generative Information Retrieval

The same idea, **generate a collection identifier rather than score candidates directly**, was popularized earlier in generative information retrieval.



- Earlier generative IR work, such as GENRE, DSI, and NCI (De Cao et al.; Tay et al.; Wang et al.), asked: *can a model generate the document identifier directly?*
- GenRec adapts this idea: the **item** plays the role of the document, and the **user history** plays the role of the query or context.
- **Shared challenge:** the identifier must be both *generatable* by the model 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).

Semantic IDs and Tokenization

Recap: How Language Models Tokenize Text

"The user watched Inception and Interstellar last night"

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

Token count: 10

The	user	watched	In	ception	and	Inter	stellar	last	night

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

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

Inception is split into `In` + `ception`*, and Interstellar into* `Inter` + `stellar`*.*

Why Use Subword Tokens?

Subwords are a compromise between words and characters.

word-level ⟷ subword-level ⟷ character-level

Q1: Why not one token per word?

- The vocabulary would become extremely large.
- Rare, misspelled, or unseen words would be hard to handle.
- New names and domain-specific terms would cause problems.

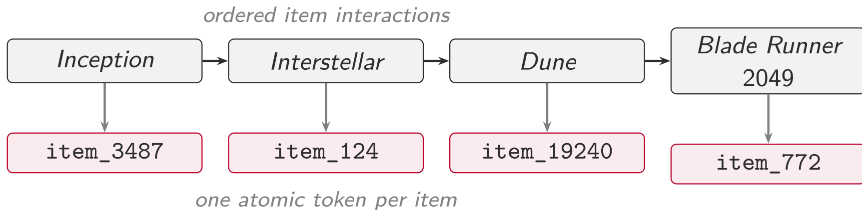
Q2: Why not one token per character?

- Sequences become much longer.
- Computation becomes more expensive.
- Long-range dependencies become harder to model.

The sweet spot: subwords keep the vocabulary manageable, reuse meaningful pieces across related words, and avoid the very long sequences produced by character-level tokenization.
Analogy for recommendation: we want item identifiers that are compact like IDs, but reusable and structured like subwords.

Item Tokenization: One Atomic Token per Item?

A user history is also a sequence: instead of words, the sequence elements are catalogue items.



Notice: Four related sci-fi films get four unrelated atomic tokens. Unless the model learns their relation from interaction data, the identifiers themselves provide no signal that they are similar. Atomic tokens are compact, but they hide useful semantic structure.

Why this is attractive

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

Why this is weak

- Vocabulary grows with catalogue size.
- Tokens are arbitrary identifiers.
- Similar items share no token structure.
- New items require new tokens and learned embeddings.

Item Tokenization: Use the Full Description as the ID?

The other extreme: make the item identifier its 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.
- Generated text may not map uniquely to one catalogue item.

Question: Can we represent each item with a few reusable tokens: shorter than full text, but more structured than one arbitrary item token?

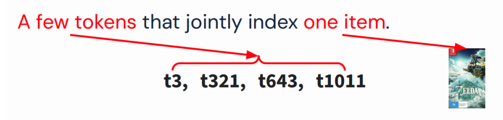
This motivates SIDs: compact token sequences that are intended to be both **generatable** and **catalogue-grounded**.

Semantic IDs: The Middle Ground

A **Semantic ID (SID)** is a short tuple of tokens/codewords that jointly identifies one catalogue item:

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

The SID tokens are **reused across items**: related items may share prefixes or individual tokens.



Source: Hou et al., CIKM 2025 tutorial.

$$\text{item A} \longleftrightarrow (t_{12}, t_{48}, t_5, t_{73})$$

$$\text{item B} \longleftrightarrow (t_{12}, t_{48}, t_{19}, t_{91})$$

The full tuple identifies the item, usually after ***collision handling****; shared prefixes or tokens can expose similarity. For example, earlier tokens may capture coarse similarity, and later tokens may refine the item identity.*

Middle ground: a SID is shorter than a full item description, but more structured than one arbitrary item token.

Why Use Multiple Tokens per Item?

Core idea: use small shared codebooks, but compose several tokens to identify each item.

Compositional capacity: with L SID positions and codebook sizes $|\mathcal{V}_1|, \ldots, |\mathcal{V}_L|$, the number of possible code sequences is

$$\prod_{\ell=1}^{L} |\mathcal{V}_\ell|.$$

If all codebooks have size $|\mathcal{V}|$, this becomes $|\mathcal{V}|^L$.

$$\underbrace{256}_{z_1} \times \underbrace{256}_{z_2} \times \underbrace{256}_{z_3} \times \underbrace{256}_{z_4} = 256^4 = 4{,}294{,}967{,}296 \approx 4.3 \times 10^9$$

This is the maximum code capacity; in practice, only assigned catalogue SIDs are valid.

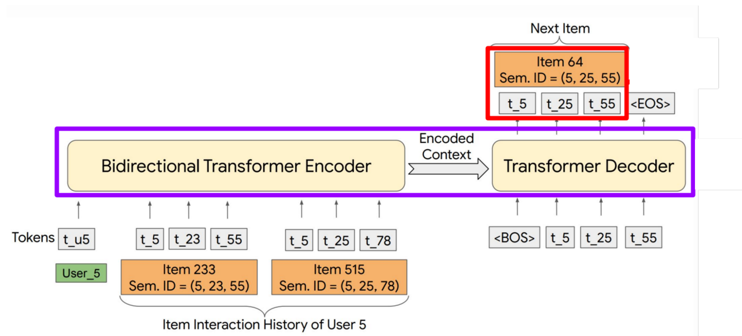
One token per item

- One vocabulary entry per item.
- Vocabulary grows with catalogue size.
- Little sharing across related items.

SID: multiple tokens per item

- Small shared codebooks.
- Full token sequence identifies the item.
- Shared prefixes/tokens can expose similarity.

SID-Based GenRec: Construction vs. Generation



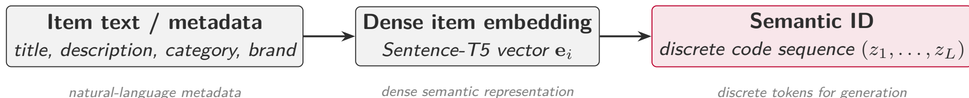
Source: Hou et al., CIKM 2025 tutorial; based on Rajput et al., *Recommender Systems with Generative Retrieval*, NeurIPS 2023.

Part 1: SID construction
How do we assign each catalogue item a short, reusable token sequence? $i \longmapsto (z_1, \ldots, z_L)$
Focus of this section.

Part 2: SID generation
How do we train a model to generate the next item's SID from user history? $p(z_\ell \mid \text{history}, z_{<\ell})$
Covered later in training and decoding.

Part 1: SID Construction in TIGER

TIGER (Rajput et al., NeurIPS 2023) is a canonical SID-based GenRec model. It first assigns each catalogue item a Semantic ID using item text and metadata.



Illustrative example

"The Legend of Zelda: Tears of the Kingdom. An epic adventure across the land . . . Video Games → Nintendo Switch → Games. Nintendo."

SID: $(7, 23, 41, 12)$

Important: this is an offline tokenization step; the recommender learns to generate these IDs later.

Inside SID Construction: RQ-VAE Quantization

Goal: assign each item a sequence of discrete codebook indices.

Step 0: trained tokenizer

The RQ-VAE tokenizer has already been trained.
Now we use it to assign IDs to catalogue items.

Step 1: encode the item

$$\mathbf{e}_i \in \mathbb{R}^{d_e}, \qquad \mathbf{r}_0 = f_\phi(\mathbf{e}_i) \in \mathbb{R}^{d_q}$$

Step 2: quantize residuals

At each level ℓ, choose the nearest vector from codebook $\mathcal{C}^{(\ell)}$.

Step 3: keep the indices

$$\mathrm{SID}(i) = (z_0, z_1, \ldots, z_{m-1})$$

At codebook level ℓ:

1. **Choose the closest codeword**

$$z_\ell = \arg\min_k \left\| \mathbf{r}_\ell - \mathbf{c}_k^{(\ell)} \right\|_2$$

z_ℓ is the **index** of the closest codeword.
This index becomes the ℓ-th SID token.

2. **Remove what was already represented**

$$\mathbf{r}_{\ell+1} = \mathbf{r}_\ell - \mathbf{c}_{z_\ell}^{(\ell)}$$

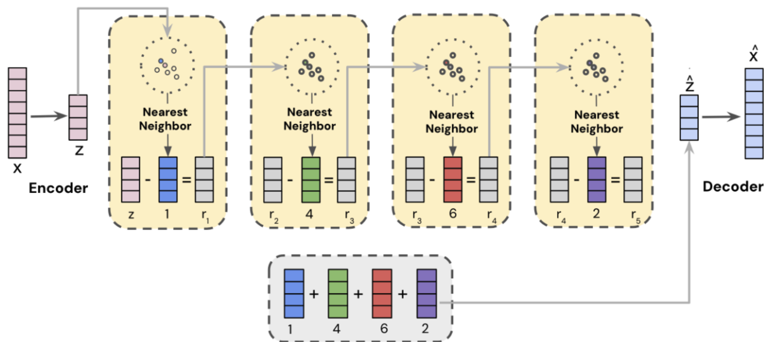
$\mathbf{r}_{\ell+1}$ is the remaining residual
passed to the next codebook.

Intuition: each codebook explains part of the item vector; the SID is the sequence of selected indices.

Inside SID Construction: RQ-VAE Quantization

train RQ-VAE tokenizer ⟶ freeze tokenizer ⟶ quantize each item ⟶ build SID lookup table

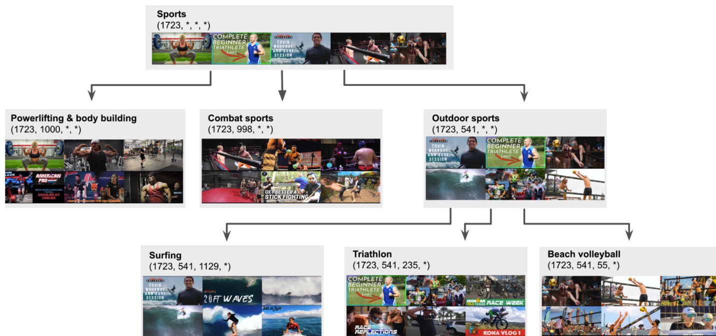
During training, the RQ-VAE encoder, decoder, and codebooks are learned to reconstruct item embeddings. After training, the tokenizer is frozen and used to assign SIDs.



Residual quantization, visual intuition. Figure source: Singh et al., *Better Generalization with Semantic IDs*, RecSys 2024.

RQ-VAE Hierarchy: Semantic Structure

RQ-VAE-based SIDs are ordered: earlier code levels give a coarse approximation of the item embedding; later code levels refine the remaining residual.



Source: Singh et al., RecSys 2024.

Related items may share early SID tokens or prefixes, giving the generator a way to share statistical signal across items instead of treating every item as a completely unrelated atomic ID.

Caveat: the hierarchy emerges from the learned embedding space and the residual-quantization scheme; it is not supervised to match human

Does the Identifier Choice Matter?

TIGER compares three item-ID choices:

- **Random identifiers**

  no semantics
- **LSH**-based SIDs

  random projections; unlearned; hash similar embeddings to similar codes
- **RQ-VAE** SIDs

  learned, hierarchical 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** +5.22%	**0.0181** +12.55%	**0.0400** +3.90%	**0.0225** +10.29%	**0.0454** +17.31%	**0.0321** +29.04%	**0.0648** +0.15%	**0.0384** +17.43%	**0.0521** +12.53%	**0.0371** +21.24%	**0.0712** +1.71%	**0.0432** +14.97%

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

Result in TIGER

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

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

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

Beyond TIGER: SID Construction Is Not Settled

GRID: practical GenRec ablations

- Re-implements SID-based GenRec in a modular framework.
- Compares tokenizers under controlled settings:
  - **RK-Means**
  - **R-VQ**
  - **RQ-VAE**
- Finds that simpler residual quantization methods can be competitive or better than RQ-VAE.

Joint Search & Recommendation

- Studies one generative model for both search and recommendation.
- Shows that the **embedding space** used to build SIDs matters.
- Reports **RQ-KMeans** as the strongest tokenizer in their joint setup.
- Task-specific IDs can help one task while hurting the other.

Takeaway

TIGER makes RQ-VAE the canonical starting point, but not the final answer. **SID construction is a modelling choice**: tokenizer, embedding space, and task objective all matter.

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.

The Design Space: How Can We Build SIDs?

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

Product Quantization
split embedding; code subspaces
PQ-style SID methods

Hierarchical Clustering
tree path IDs; coarse → leaf
hierarchical item IDs

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

Trade-offs:

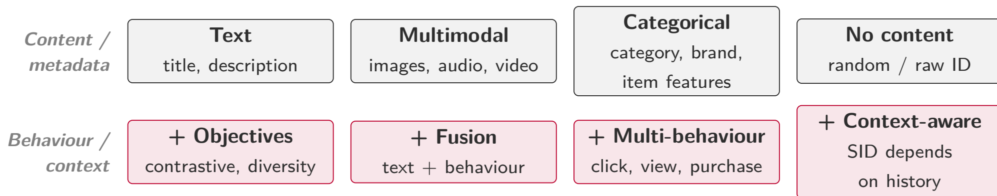
semantic content vs. **behaviour alignment** vs. **validity and decoding efficiency**

Main message

Semantic ID design is an **active modelling choice**: the tokenizer, the embedding space, and the downstream task all affect performance.

What Information Shapes the SID?

SID quality depends not only on the tokenizer, but also on the **signals** and **objectives** used to build the item representation.



The trend

From **static, content-only** IDs → **behaviour-aware, context-aware, task-aware** IDs.

Related message: SID performance depends on tokenizer choice, embedding space, and downstream objective.

CoST: Better Tokens via Contrast

RQ-VAE-style tokenization:

$$\mathbf{e}_i \longrightarrow \hat{\mathbf{e}}_i$$

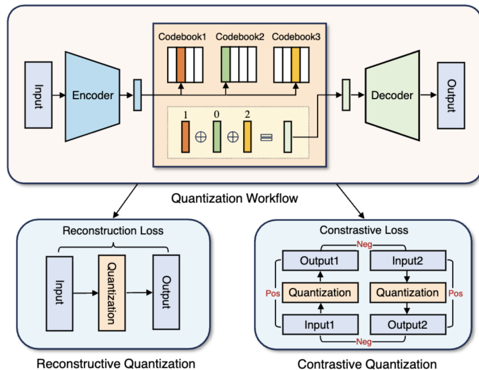
Limitation: reconstruction focuses on each item embedding individually. It does not explicitly preserve which items are close neighbours.

CoST adds contrastive quantization:

$$\mathrm{sim}(\mathbf{e}_i, \hat{\mathbf{e}}_i) > \mathrm{sim}(\mathbf{e}_i, \hat{\mathbf{e}}_j), \quad j \neq i$$

Reported result on MIND:

$$+43\% \text{ Recall@5}, \quad +44\% \text{ NDCG@5}$$



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

Intuition

The tokenizer should preserve item **neighbourhoods**, not just reconstruct individual vectors.

LETTER: Behaviour-Aware Tokenization

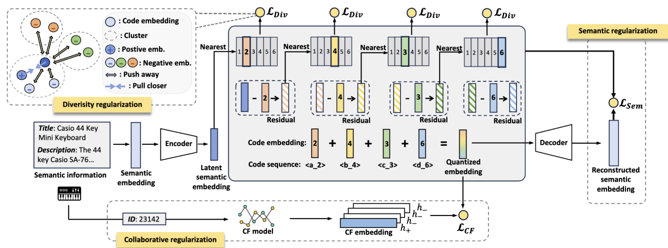
Motivation: similar item *text* does not always imply similar user *behaviour*.

LETTER adds three regularizers:

- **Semantic:** preserve hierarchical item meaning.
- **Collaborative:** align codes with CF signals,

  i.e., user–item interaction patterns such as signals from user behaviour, such as which items are clicked, bought, watched, or co-consumed by similar users.
- **Diversity:** reduce code assignment bias.

Can be used as a tokenizer replacement in SID-based GenRec models.



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

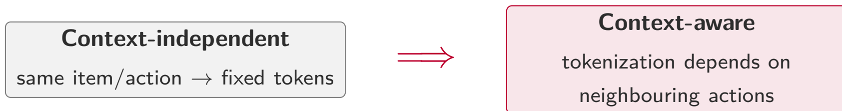
Lesson: good item tokens should capture semantics, behaviour, and balanced code usage.

Frontier: Context-Aware Tokenization

Most SID-based GenRec methods tokenize each item/action **independently**: the same item gets the same fixed token sequence wherever it appears.

ActionPiece (Hou et al., ICML 2025) changes this:

- Each action is represented as an **unordered set of item features**.
- A WordPiece-like procedure merges frequent feature patterns into new tokens.
- Merges can happen **within an action** or **across adjacent actions**.
- Therefore, the same action can be tokenized differently depending on its surrounding context.



Main idea

Move from **fixed item identifiers** to **learned action tokens** that can encode feature patterns from the item itself and from its local sequence context.

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

Tokenization: Three Things to Remember

1. Why?

Items are not naturally tokenized like words.

GenRec must define an **output space**.

2. What?

A **Semantic ID** is a short shared code:

$$i \mapsto (z_1, \ldots, z_L)$$

3. How?

SIDs depend on the tokenizer, input signals, and objective.

RQ-VAE is one option, not the final answer.

Main lesson: Tokenization is not preprocessing. It defines what the generator can learn, generate, and map back to the catalogue.

text-only → **multimodal** → **behaviour-aware** → **context-aware**

Next

Now that items have Semantic IDs, how does the model *generate* the next item's ID from a user history?

Training and Decoding

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.

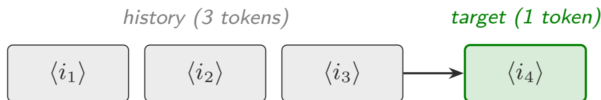
Building Training Examples — Atomic IDs

Input: chronological user interaction sequence.

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

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.

Architecture: Encoder–Decoder or Decoder-Only?

Encoder–decoder

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

- *Encoder reads the full history.*
- *Decoder emits the next SID.*
- *Good when history is bounded.*

Decoder-only

GPT-style; used by HSTU, OneRec, GPTRec

- *Single autoregressive stream:*

  [*history* || *target SID*]

- *Scales better with very long histories.*
- *Industrial trend (2024–2025).*

Both produce SID tokens autoregressively:

$$p(z_1, \ldots, z_L \mid h) \;=\; p(z_1 \mid h) \prod_{\ell=2}^{L} p(z_\ell \mid h,\, z_{<\ell})$$

Takeaway

The model is a **sequence model**. Everything you know about Transformers applies, attention, positional encoding, scaling. The only new thing is what the tokens *mean*.

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**.

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.

Beyond Cross-Entropy: Reward-Based Fine-Tuning

Limits of next-token CE: only matches the exact historical next item, no signal for **validity**, **listwise quality**, or business goals (diversity, freshness, safety).

Fix: after CE pre-training, fine-tune on **full generated SIDs** with an RL objective. **GRPO** (Group Relative Policy Optimization, DeepSeek) is the default: PPO without a value net, advantages come from comparing each sample to its group mean.

Loop (per history):

1. Sample G candidate SIDs from π_θ.
2. Score: $r_g = R(o_g \mid \text{history})$.
3. Advantage $A_g = (r_g - \bar{r})/\sigma_r$.
4. PPO-clipped update + KL to π_{ref}.

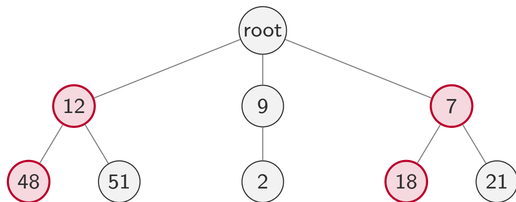
Reward components:

- **Validity**: SID exists in catalogue trie.
- **Relevance**: ground-truth match / reward model.
- **Diversity**: penalize prefix overlap in group.
- **Business**: freshness, margin, safety.

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:

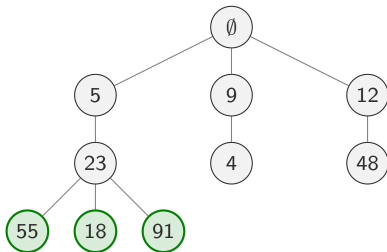
- The SID space has K^L possible codes (e.g., $\sim 10^9$).
- The catalogue uses only a tiny fraction (e.g., $\sim 10^7$ items).
- Most code combinations are **unused**.

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.

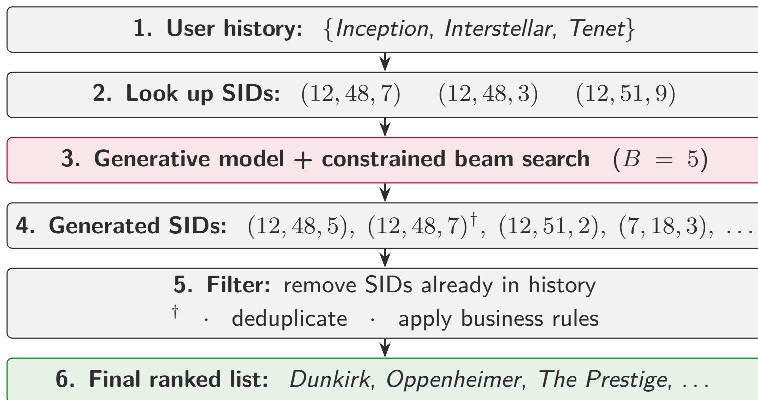
Effect:

- 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, non-trivial in fast-moving systems.

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.*

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 $(z_1^\star, \dots, z_L^\star)$.
- 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.

Caveat

The model can now *emit* $i^\star$, but won't until its prefix becomes likely. **Exploration** (sampling, ϵ-greedy, RL with novelty reward) is what actually surfaces cold items.

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.

Homogeneity

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

Local optima

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

Inference cost

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

Takeaway

Decoding is part of recommender design, not an inference detail. We come back to this in the next section.

Fixing Homogeneity: Diverse Decoding

Goal: keep validity (trie) and quality, but spread the beam across **different regions** of SID space.

Inference-time fixes

- **Diverse beam search**: split B beams into G groups; penalize groups for token overlap with earlier groups (Vijayakumar et al., 2016).
- **Nucleus / top-p sampling**: sample from the smallest set whose mass $\geq p$. Trades off determinism for spread.
- **Temperature**: $\tau > 1$ flattens the distribution; popular SIDs lose dominance.
- **MMR re-rank**: pick top-1, then iteratively pick the candidate that maximizes relevance − similarity to already-picked.

Training-time fixes

- **Diversity reward in RL** (Frame 3b): penalize prefix overlap inside the GRPO group.
- **Tokenizer-level** (LETTER): *diversity loss* on code assignment prevents popular items from collapsing onto the same prefix (arXiv:2405.07314).
- **Determinantal** / submodular objectives over slates.

Trade-off

More diversity ⇒ lower top-1 accuracy on *historical* next item, but typically higher **coverage**, **novelty**, and online engagement. The right setting 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 ($\sim$50) — TIGER \| Long ($\sim$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

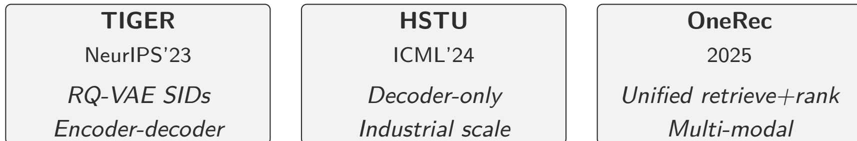
Recap

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

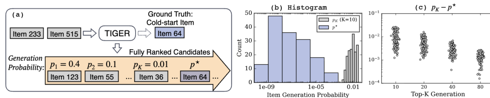
Harder #1: Cold-Start Becomes Fragile

Recall (Sec. 4): a new item becomes **decodable** the moment we tokenize it. But *decodable* $\neq$ *recommended* : and this is where the promise breaks.

Surprise: despite the cold-start *promise* of SIDs, generative retrieval can do **worse** than dense retrieval on cold-start.

Why this happens:

- During training, the model only sees **SIDs of frequently-clicked items**.
- Even if a cold item has a valid SID, its generation probability is too low.
- Beam search prunes it before it can be considered.



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

LIGER's fix:

generate $\rightarrow$ augment with cold items $\rightarrow$ dense re-rank

Lesson

The future of GenRec may be **hybrid**, not purely generative.

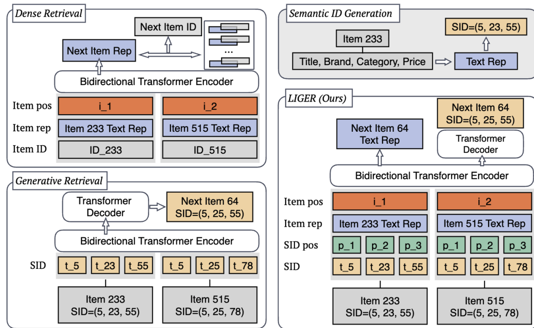
Harder #2: Dense Retrieval Is Still Strong

Dense retrieval

- One embedding per item.
- Strong ranking, simple inference.
- Cold-start handled via text.
- But: storage + ANN search at scale.

Generative retrieval

- Compact SID representation.
- Efficient candidate generation.
- But: cold-start fragility.
- And: ranking quality varies.



Source: Yang et al., 2024.

Academic vs. Industrial view

On **academic benchmarks** (Amazon Reviews, MovieLens), dense retrieval often wins.
At **industrial scale** (HSTU, OneRec), generative recommenders outperform production DLRMs.

The winning paradigm depends on regime, not just on algorithm.

Harder #3: Decoding Is Expensive and Biased

Recall from Section 4 : beam search over SIDs has GenRec-specific problems:

Amplification bias

Popular SID prefixes dominate; tail items pruned early.

Homogeneity

Top-B candidates share prefixes : low diversity.

Inference latency

L autoregressive steps per recommendation. Hard to meet $<$50ms SLA at industrial scale.

Trie maintenance

Constrained decoding requires keeping the trie in sync with a fast-changing catalogue.

Active research:

- **Speculative decoding** (Leviathan et al. '23) : cheap draft model proposes, expensive model verifies.
- **Parallel SID generation** : RPG (KDD'25) emits unordered codes in parallel.
- **Prefix / KV caching** over the trie : popular SID prefixes are recomputed millions of times.
- (Diversity-aware decoding: covered in Sec. 4, Frame 8b.)

Harder #4: Catalogues Move; Evaluation Lags

Dynamic catalogues

- New items every minute (TikTok, news, e-commerce).
- Each needs a SID : can we assign one without retraining the quantizer?
- Old items disappear : does the model forget gracefully?
- **Knowledge editing for RecSys** is a fresh open problem.

Evaluation mismatch

- Standard offline metrics: Recall@K, NDCG@K, MRR@K.
- But generation enables *novel* candidates not in the test set.
- Static benchmarks under-credit useful generation.
- Real systems also need: diversity, novelty, fairness, long-term engagement.

Open question

What's the right benchmark for a model that doesn't just *rank*, but *generates*?

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

A cascade 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** : but the safety tooling is still cascade-shaped.

What Becomes Possible

New Capabilities Unlocked by Generation

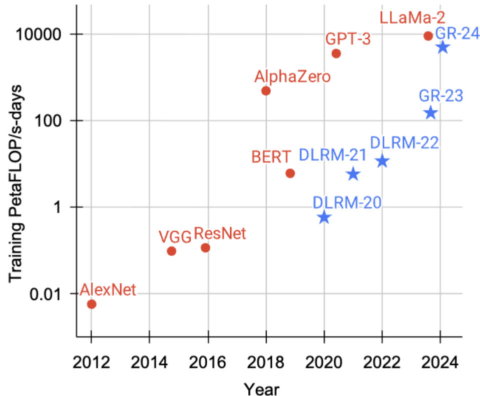
Possible #1: Scaling Laws for Recommendation

The question: can RecSys models scale like LLMs?

Actions Speak Louder than Words (ICML'24):

- User actions as a **new modality**.
- **HSTU**: long-sequence architecture for high-cardinality recommendation.
- **M-FALCON**: efficient target-aware inference.

Empirical result: GR scales to LLM-level training compute and outperforms production DLRMs.



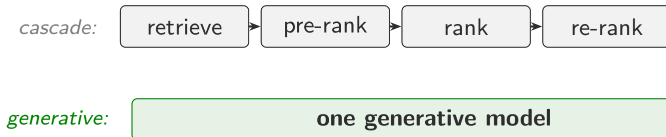
Source: Zhai et al., ICML 2024.

The shift

From task-specific recommenders → **large generative recommender models** trained on user actions.

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.

Possible #4: Test-Time Reasoning for RecSys

LLMs got dramatically better at **reasoning** by spending more compute at test time (chain-of-thought, o1-style models). Can RecSys do the same?

Think Before Recommend (Tang et al., 2025) : multi-step reasoning over latent hidden states:

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

Reported gains (NDCG@20):

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

(Upper bounds with oracle reasoning: +37% to +53%.)

Open question

Is RecSys having its **reasoning moment**?
If yes, the next 2–3 years of GenRec research may look very different from the last 2–3.

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

Key Takeaways

1. Generative recommendation reframes recommendation as **sequence generation**, from $s(u, i)$ to $p(z_1, \ldots, z_L \mid h)$.
2. **Semantic IDs** are the recommendation analogue of subword tokens: short, shared, structured codes that sit between raw IDs and full text.
3. Construction (RQ-VAE & variants) is a **modelling choice**, not preprocessing, it bounds everything downstream.
4. Training is standard next-token cross-entropy; **decoding** (trie-constrained beam search, diversity-aware variants) is part of the *model*, not inference plumbing.
5. Today: dense retrieval remains **competitive on small academic benchmarks**, but the gap shrinks with better tokenizers (LETTER, EAGER, RPG); generative models win at **industrial scale**. *Hybrid* approaches (LIGER) bridge the gap.
6. Safety, privacy, and auditability are **LLM-shaped problems** that GenRec inherits : the trie and the tokenizer are now part of the safety surface.
7. Tomorrow: **foundation models for behaviour**, unified retrieve+rank, and test-time reasoning.

Discussion Questions

1. Why is generative recommendation an *interesting* problem, beyond being a new technique? What does the reformulation buy you?
2. Semantic IDs were motivated by analogy to BPE. Where does the analogy hold, and where does it break down?
3. LIGER suggests that pure generative retrieval may not always be the right answer. When *should* you reach for a generative model?
4. Test-time reasoning gives +9% on SASRec but the oracle upper bound is +53%. What does this gap tell us?
5. If you were starting a PhD in this area tomorrow, which of the open challenges would you pick? Why?