

TGR:

Advancing Industrial Recommendation from Generative-Paradigm Ranking toward Unified Generation and Reasoning

TGR Team

Industrial recommender systems have long followed a cascaded architecture—retrieval, pre-ranking, ranking, and reranking—powered by separately optimized Deep Learning Recommendation Models (DLRMs). This blueprint now faces three structural challenges. First, fragmented DLRM architectures process high-cardinality, heterogeneous features through separately designed modules and exhibit no LLM-style scaling law: quality saturates as model capacity and compute grow. Second, stage-wise optimization and progressive candidate truncation fragment the decision process, and pointwise item scoring ignores the inter-item dependencies and positional effects that determine whole-slate quality. Third, co-occurrence-driven, ID-based models lack the multimodal semantics, world knowledge, and deliberate reasoning needed for cold-start, long-tail, and ambiguous-intent requests. Together, these challenges are driving an industry-wide shift from the cascaded toward the generative paradigm: unified, scalable sequence modeling that unlocks predictable scaling, end-to-end generation that decides the slate as a whole, and LLM-derived priors that supply world knowledge and reasoning.

To address these challenges across the heterogeneous business scenarios within Tencent’s PCG (Platform and Content Group), spanning diverse content modalities, user populations, candidate-set sizes, and latency requirements, we develop **TGR (Tencent Generative Recommendation)**, an industrial framework that advances recommendation from conventional cascades toward the generative paradigm along three coupled directions. First, **TGR-GenRank** upgrades the ranking stage to generative-paradigm ranking. Its production model, CCFormer, combines unified feature tokenization and a scalable Transformer backbone with feature-field separated cross attention, long-sequence subspace token mixing, and hierarchical sequence compression, while retaining per-item multi-task scores for drop-in deployment. Second, **TGR-GenRec** explores end-to-end replacement of the cascade’s decision stages through two complementary generation paradigms. Under next-token prediction (NTP), BARGE formalizes and bridges two structural gaps in hierarchical semantic-ID generation—the loss of item boundaries when multi-token items are flattened and semantic drift caused by early prefix errors—through item context-aware attention, hierarchical path reranking, and orthogonal dual-path decoding. Under next-slate prediction (NSP), HiGR reformulates recommendation from item-by-item decoding to whole-slate generation and co-designs prefix-structured semantic IDs for controllable planning, a coarse-to-fine decoder that separates global slate planning from local item generation, and listwise multi-objective alignment over ranking fidelity, genuine user interest, and diversity. Third, **TGR-Reason** augments end-to-end generation with LLM reasoning amortized through latent multi-step supervision. Offline, a LatentRec-trained Think model materializes complete semantic-ID *reason tokens*; Direct Reasoning Injection uses them as hierarchy-aligned prompts to condition online decoding without a request-time reasoning rollout.

TGR is deployed at scale on Tencent production traffic, serving hundreds of millions of users across surfaces including news, video, and web novels. CCFormer outperforms HSTU, One-

Trans, and STCA on public benchmarks and a 4-billion-sample production dataset, scales predictably at roughly half the GFLOPs of HSTU, trains $2.21\times$ faster, delivers statistically significant lifts in all five A/B-tested production scenarios, and is fully launched in two of them (+3.57% CTR in the video-recommendation scenario; +1.71% advertising revenue in the advertising-ranking scenario). BARGE outperforms OneRec by 10.2–16.9% in Hit@5 across two industrial scenarios and is fully rolled out as a generative retrieval channel with parameters, beam width, and serving latency unchanged, delivering +0.60% CTR and +1.70% total reading time—gains that grew larger and more stable after full launch—and the highest CTR among the platform’s representative retrieval channels. HiGR improves offline slate quality by 15.9–21.3% over matched-capacity OneRec with a $5\times$ inference speedup, serves whole slates below 50 ms P99 latency at roughly 60% lower GPU demand than full-sequence decoding, and consistently lifts engagement, traffic consumption, and commercial value across three A/B-tested scenarios (up to +1.22% Average Watch Time and +1.73% Average Video Views, +0.68% CTR, +0.56% advertising revenue). Offline, TGR-Reason raises cold-start new-user Hit@1 by +477.8%; on-line, it delivers statistically significant gains of +1.75% in Effective Consumption Rate and +13.09% in new-user Exposure-to-Conversion Rate.

Contents

1	Introduction	3
2	The TGR Stack: Design Principles and Shared Foundations	6
2.1	Design Principles	6
2.2	Problem Formulation	8
2.3	Shared Components	8
3	TGR-GenRank: Generative-Paradigm Ranking	8
3.1	Motivation	9
3.2	CCFormer Architecture in Brief	10
3.3	Training and Serving at Industrial Scale	11
3.4	Offline Experiments	12
3.5	Scaling Behavior	13
3.6	Online Deployment and A/B Results	14
4	TGR-GenRec: From Next-Token to Next-Slate Generation	16
4.1	Two Generation Paradigms: Next-Token and Next-Slate Prediction	16
4.2	BARGE: Adapting Next-Token Prediction for Industrial Recommendation	17
4.2.1	Motivation: Two Structural Gaps at Industrial Scale	17
4.2.2	Method Overview	19
4.2.3	Offline Results	21
4.2.4	Efficiency and Engineering Considerations	22
4.2.5	Deployment and Online A/B	23
4.3	HiGR: Slate Generation for Industrial Recommendation	24
4.3.1	Motivation: Three Disconnects Between Item Generation and Slate Optimization	24
4.3.2	Method Overview	25
4.3.3	Offline Results	26
4.3.4	Efficiency and Engineering Considerations	27
4.3.5	Deployment and Online A/B	29

5	TGR-Reason: Reasoning-Augmented Generative Recommendation	30
5.1	Motivation	30
5.2	Method Overview	31
5.2.1	Reason Tokens as a Shared Reasoning–Memory Interface	31
5.2.2	Offline Reason Token Generation with LatentRec	32
5.2.3	Reasoning and Memory Injection into TGR-GenRec	34
5.3	Training Recipe	36
5.3.1	Think-Model Training	36
5.3.2	Gen-Model Training	37
5.3.3	Offline Generation at Scale	37
5.4	Deployment	38
5.4.1	Serving Efficiency	38
5.4.2	Online Serving	40
5.4.3	Store Consistency and Failure Isolation	41
5.5	Evaluation Results	42
6	Conclusion, Limitations, and Future Directions	43
6.1	Conclusion	43
6.2	Limitations	43
6.3	Future Directions	44
	Appendix	44
A	Contributions	44
B	Notation	45
C	Default Configurations and Hyperparameters	45
D	Complexity Analysis	45
	References	48

1. Introduction

For a decade, industrial recommender systems have shared a common blueprint: a cascaded architecture of retrieval, pre-ranking, ranking, and reranking, in which the major decision stages are powered by separately optimized Deep Learning Recommendation Models (DLRMs) [1–5]. The blueprint is modular and operationally convenient, and it has carried industrial recommendation through years of continuous evolution. Yet it is now facing three structural challenges at once—challenges that, we argue, cannot be tuned away from inside the blueprint:

- (i) **The ranking architecture no longer converts compute into quality.** A production DLRM processes high-cardinality, heterogeneous features through separately designed modules for feature extraction, feature interaction, and representation transformation [6]. This fragmented architecture is memory-bandwidth-bound, sustains low Model FLOPs Utilization, and exhibits no LLM-style scaling law: added parameters no longer buy quality, and feature interactions are bounded by what can be hand-engineered [7]. Generative-paradigm rankers offer the first credible break through this ceiling—by tokenizing user, item, context, and sequence features into a single token stream and letting a large Transformer do the crossing, HSTU scales sequential transducers into the trillion-parameter regime and ships +12.4% in online A/B at Meta [6], while OneTrans lifts per-user GMV by +5.68% in production at strict latency budgets [8]—but their self-attention cost grows

Table 1 | TGR vs. representative industrial generative recommenders. “Item-AR” = item-level autoregressive decoding; “Slate-AR” = list-level autoregressive decoding; “CoT” = chain-of-thought reasoning; “reason tokens” = reasoning-enriched semantic IDs generated offline by a LatentRec-trained Think model (§5.2.2); “PRL” = per-step reasoning loss.

System	Stage replaced	Tokenizer	Decoder	Alignment	Reasoning
HSTU [6]	Retrieval / ranking	Sequence ID	Sequential transducer	Multi-task	—
OneTrans [8]	Ranking	Unified tokens	Causal Transformer	Pointwise	—
OneRec-V1 [16, 17]	Retrieval + ranking	RQ-Kmeans	Enc-Dec, Item-AR	DPO	—
OneRec-V2 [18]	Retrieval + ranking	RQ-Kmeans	Lazy Decoder, Item-AR	GBPO + user feedback	—
OneRec-Think [10]	Retrieval + ranking	RQ-Kmeans	Lazy Decoder, Item-AR	RL + In-text CoT	In-text CoT
OneReason [20]	Retrieval + ranking	RQ-Kmeans	LLM decoder (Qwen3), Item-AR	CoT SFT + RL	In-text CoT (3-level)
GPR [19]	Full pipeline (ads)	Unified rep.	HHD	VAFT + HEPO (RL)	—
PinRec [21]	Retrieval	Multi-token	Multi-token AR	Outcome-cond.	—
TGR-GenRank (CCFormer)	Ranking	Unified tokens	Cross-field attn. + compr.	Multi-task	—
TGR-GenRec (BARGE)	Retrieval	OSQ-VAE (dual)	Dual-decoder, Item-AR + HPR	Aux. InfoNCE	—
TGR-GenRec (HiGR)	Retrieval / Retrieval + ranking	PCRQ-VAE	Hierarchical, Slate-AR	ORPO	—
TGR-Reason	Retrieval / Retrieval + ranking	Shared SID (PCRQ-VAE)	Slate-AR + reason tokens	NTP + PRL	Latent (offline)

quadratically with behavior-sequence length, so fine-grained cross-field interaction is routinely sacrificed, via sequence pre-compression or truncation, to contain cost.

- (ii) **Stage-wise, item-wise decisions misalign with list-level experience.** Stage-wise optimization and progressive candidate truncation fragment the decision process, causing information loss and objective mismatch across the cascade. The fragmentation runs deeper than the pipeline: what a user actually experiences is an ordered *slate* in a single display, whose quality hinges on the inter-item dependencies and positional effects that pointwise, item-in-isolation scoring cannot capture [9]—and the direct generative remedy, decoding an M -item slate of D -token semantic IDs autoregressively, is too slow under real-time constraints.
- (iii) **Pattern matching is not reasoning.** Conventional recommender models rely heavily on behavioral co-occurrence patterns learned from historical user-item interactions and encoded in ID-based representations. Their limited access to multimodal item semantics, world knowledge, and deliberate reasoning constrains the inference of nuanced or ambiguous user intents and generalization under sparse supervision—precisely the cold-start, long-tail, and ambiguous-intent regimes in which recommendation quality is decided. LLM-derived priors offer the natural remedy. In existing approaches, however, the reasoning is executed for every request or refresh—as chain-of-thought text [10] or as multi-step latent rollouts [11]—a cost that industrial latency budgets cannot absorb on the serving path and that offline precomputation only relocates.

Catalyzed by the broader success of generative AI [12–15], the shift has unfolded along three axes: unified, scalable sequence modeling, carried by generative-paradigm rankers such as HSTU [6] and OneTrans [8]; end-to-end generation, where Kuaishou’s OneRec series [16–18] replaces the retrieve-rank pipeline with a single generator at 0.1B–8B parameters, while GPR [19] pursues a unified “one-model paradigm” for advertising; and LLM-derived reasoning, where OneRec-Think [10] adds in-text reasoning to OneRec, and OneReason [20] achieves a thinking mode that outperforms the non-thinking mode on Kuaishou business benchmarks. Together, these systems establish that the cascade can be revisited piece by piece or replaced outright; they are the systems TGR benchmarks against or is positioned alongside in Table 1.

Our setting. Within Tencent’s Platform and Content Group (PCG), recommendation runs across a deliberately heterogeneous portfolio of surfaces—news feeds, short videos, long videos, music, and web novels—which together serve hundreds of millions of daily active users, with candidate corpora of tens to hundreds of millions of items and tight per-stage P99 latency budgets in the tens of milliseconds.

The TGR Stack. To address these challenges in this setting, we develop **TGR (Tencent Generative Recommendation)**, an industrial framework that advances recommendation from the cascaded blueprint toward the generative paradigm along three coupled directions—the **TGR Stack** (Figure 1):

1. **TGR-GenRank — generative-paradigm ranking (§3).** Building on the HSTU [6] and OneTrans [8] line of work, TGR-GenRank keeps the inherited cascade in place and replaces the production rankers with **CCFormer** [22], an efficient Transformer backbone that unifies cross-field feature interaction and compressed long-sequence modeling: feature-field separated cross attention lets user, behavior-sequence, and target-item tokens interact along semantically directed flows, while subspace token mixing and hierarchical sequence compression make long-sequence modeling subquadratic without giving up access to the full history. Trained with unified multi-task supervision on LLM-style infrastructure, CCFormer is deployed as a drop-in replacement for the production ranker in two scenarios. Note that, although TGR-GenRank adopts the generative paradigm’s tokenized representation, Transformer backbone, and scaling recipe, the ranking output remains per-item scores—it *borrow*s the generative paradigm and is therefore best understood as “generative-paradigm ranking”, not yet “end-to-end generation”.
2. **TGR-GenRec — unified end-to-end generative recommendation (§4).** The direction that breaks out of the cascade entirely: its goal is to replace retrieval, ranking, and reranking with a single generative model that decides the result list end to end, removing per-stage information loss and objective drift. Aligned with the one-model direction of OneRec at Kuaishou, TGR-GenRec currently ships one production model per generation paradigm (§4.1): **BARGE** [23], which structurally adapts the next-token-prediction (single-item) generation mechanism to the recommendation task through item context-aware attention, hierarchical path reranking, and orthogonal dual-path decoding; and **HiGR** [24], which moves up to whole-slate generation—differentiating from OneRec-V1/V2 by emitting the entire *slate* as one structured object rather than decoding items one at a time—through Prefix-Contrastive RQ-VAE tokenization (PCRQ-VAE), a Hierarchical Slate Decoder (HSD), and ORPO-based listwise multi-objective alignment.
3. **TGR-Reason — reasoning-augmented generative recommendation (§5).** Among the possible carriers for connecting LLM reasoning with a generative recommender—chain-of-thought text, auxiliary generated features, latent reasoning states, and reasoning-enriched semantic IDs (§5.2.1)—TGR-Reason adopts the last: a Think model trained with LatentRec internalizes latent reasoning at training time and periodically exports *reason tokens* offline, which are injected into the online generator (and can further query Group Memory) without adding any reasoning rollout to the request path. In the FM4RecSys 3-paradigm taxonomy (Feature-Based → Generative → Agentic) [25], TGR-Reason is best understood as the on-ramp from *Generative* toward *Agentic*: it keeps a single generator but equips it with deliberate reasoning, while leaving room for tool use, multi-turn dialogue, and autonomous agents—hallmarks of the fully agentic paradigm—in later iterations.

Contributions. All three directions are deployed on our production traffic and validated in online A/B tests; this report consolidates the design decisions and the offline, scaling, ablation, and online evidence behind each. Its contributions are:

1. We describe an industrially deployed generative recommendation stack (TGR) spanning three coupled directions—TGR-GenRank, TGR-GenRec, and TGR-Reason—under a common production discipline, with a semantic-ID tokenizer family, user/context encoder, and post-training pipeline shared across the generative-recommendation models. TGR currently serves hundreds of millions of users across multiple Tencent surfaces.

2. We provide a detailed technical description of **TGR-GenRank**’s production model **CCFormer**—feature-field separated cross attention, long-sequence subspace token mixing, and hierarchical sequence compression—with an emphasis on its industrial training and serving recipe (mixed-precision training, INT8 + double-hashing sparse-parameter compression, and single-pass parallel candidate scoring on the Numerous-Torch infrastructure), validated across five production scenarios, two of which are fully launched.
3. We provide a detailed technical description of **TGR-GenRec**, covering one production model per generation paradigm: BARGE, which structurally adapts next-token-prediction SID generation via item context-aware attention, hierarchical path reranking, and dual-path decoding; and HiGR, which realizes whole-slate generation through its tokenizer (PCRQ-VAE), coarse-to-fine decoder (HSD), and ORPO-based listwise multi-objective alignment—together with offline and online results for both.
4. We present the design and deployment of **TGR-Reason**: a LatentRec-trained offline Think model that amortizes reasoning at training time and exports reasoning-enriched semantic-ID priors (*reason tokens*), injected into the production generator at zero request-path reasoning cost. On one of our commercial content platforms it delivers consistent offline gains, strongest for cold-start new users, and statistically significant online A/B lifts.

The remainder of this report is organized as follows. We first introduce the TGR Stack and shared components (§2). We then describe TGR-GenRank (§3); the TGR-GenRec direction (§4), whose two production models are presented as independent, parallel works—BARGE (§4.2) and HiGR (§4.3); and TGR-Reason (§5). Each model chapter reports its own deployment and online A/B results in place; Section 6 concludes with limitations and future directions. The author list is provided in Appendix A, a notation table in Appendix B, consolidated default configurations in Appendix C, and complexity analyses in Appendix D.

2. The TGR Stack: Design Principles and Shared Foundations

This section describes the design principles that govern TGR as a whole and the components that are shared across TGR-GenRank, TGR-GenRec, and TGR-Reason. Figure 1 shows how the pieces fit together: a ranking layer that upgrades the cascade in place, a generation-and-reasoning layer that replaces it end to end, and a foundation layer—tokenizers, encoders, and training infrastructure—on which both build.

2.1. Design Principles

Three principles guide the entire TGR Stack:

(P1) Replace the cascade piece by piece, each piece industrial-grade. A useful generative recommender is one that can replace a heavily tuned discriminative module *at full production quality* from day one. Each TGR model is therefore deployed as a drop-in replacement for a specific stage, behind the existing safety, dedup, and business-rule filters, and is held to two bars: (i) match or exceed the offline metrics of the incumbent, and (ii) stay within the same online latency and GPU budgets. This discipline recurs throughout the report: CCFormer ships as a drop-in ranker on the production feature space (§3.3), BARGE was deployed under a hard no-increase constraint on parameters, beam width, and serving latency (§4.2), and HiGR serves whole-slate generation within a strict latency-and-GPU envelope (§4.3.4).

(P2) Share structure across the stack. Generative recommendation works at scale only if tokenizer, user encoder, and post-training infrastructure can be amortized across models. TGR

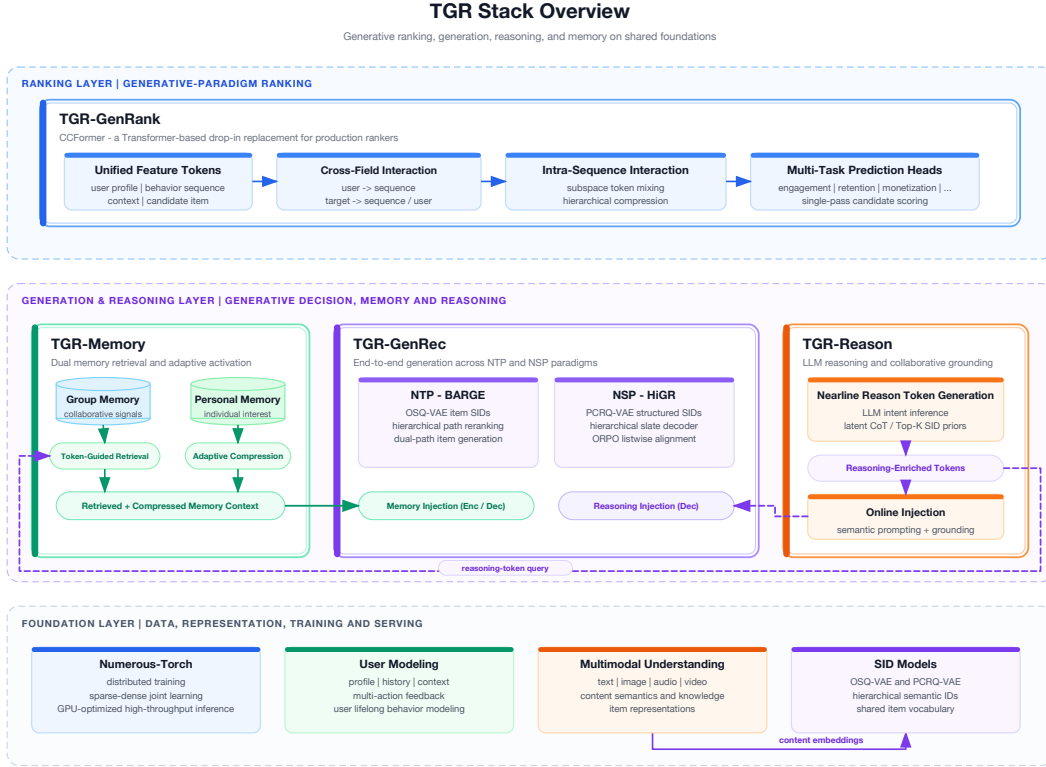


Figure 1 | Overall architecture of the TGR Stack. *Top*: the ranking layer—TGR-GenRank (CCFormer) turns unified feature tokens into per-item multi-task scores through cross-field and intra-sequence interaction, with single-pass candidate scoring. *Middle*: the generation and reasoning layer—TGR-GenRec generates the result list end to end under the NTP (BARGE) and NSP (HiGR) paradigms, with TGR-Reason injecting reasoning-enriched tokens and TGR-Memory supplying retrieved memory context (the Group Memory retrieval path is a design extension, excluded from the deployed configuration; §5.2.3). *Bottom*: the foundation layer—Numerous-Torch training and serving, user modeling, multimodal understanding, and the SID models (OSQ-VAE and PCRQ-VAE) shared across the stack.

therefore enforces a single semantic-ID (SID) tokenizer family (PCRQ-VAE for slate generation and the orthogonal dual-channel OSQ-VAE for NTP generation; §4.3.2, §4.2.2), a common user/context encoding interface, and a common post-training discipline. Sharing the SID space is what lets TGR-Reason’s reason tokens be injected natively into TGR-GenRec’s decoder (§5.2.3), and what will allow a future generative head for TGR-GenRank to be bootstrapped from TGR-GenRec’s fine-grained item generator.

(P3) Optimize for the list, not the item. The unit of user experience on our surfaces is the displayed list, so the stack-level objective is defined over whole slates rather than isolated items. Each stage honors that objective at the level its paradigm permits: online, every model is gated on list- and session-level business outcomes; the slate-generation line trains and aligns directly at the list level, integrating ranking fidelity, genuine user interest, and diversity in a single listwise ORPO loss over slate-level preference pairs (§4.3.2) instead of fusing per-item rewards by hand; and the NTP line and the ranking stage supply the item-level fidelity that the list objective consumes (§4.1). This list-first commitment is consistent with the evidence that reward design often matters more than raw scale [18] and with the emphasis on session-level objectives in scaled sequential transducers [6].

2.2. Problem Formulation

Let $\mathcal{U}$ denote the set of users and $\mathcal{V}$ the set of items. For each user $u \in \mathcal{U}$, we observe a chronological interaction history $S^u = (v_1^u, v_2^u, \dots, v_n^u)$ together with a set of user, context, and request-level features x^u . On each request, the system returns an ordered slate of M items

$$O^u = (\hat{v}_1^u, \hat{v}_2^u, \dots, \hat{v}_M^u),$$

optimized for holistic list-level user feedback (e.g., watch time, app stay time, completion, satisfaction) rather than for per-item scores in isolation.

TGR pursues this objective in two deployment regimes. *Within the cascade*, the slate is still assembled by the downstream stages, and TGR-GenRank upgrades the ranker that drives them: CCFormer consumes the production feature space and emits per-item multi-task scores for all candidates of a request in a single pass (§3). *End to end* (TGR-GenRec and TGR-Reason), a generative Transformer $\mathcal{F}_\theta$ decodes the slate directly as a sequence of semantic-ID tokens,

$$O^u = \mathcal{F}_\theta(u, S^u, x^u),$$

with TGR-Reason conditioning the same decoding on reasoning-enriched priors (§4, §5). The full notation is summarized in Appendix B.

2.3. Shared Components

Semantic-ID tokenizers. The slate-generation models (HiGR and TGR-Reason) share the same tokenizer, PCRQ-VAE (§4.3.2). BARGE, under the NTP paradigm, uses the orthogonal dual-channel OSQ-VAE (§4.2.2). In either case, each item v is mapped to a hierarchical SID sequence $(s^1, s^2, \dots, s^D)$ in which high-level prefixes encode coarse semantic and collaborative structure and the terminal layer disambiguates individual items. TGR-GenRank sits outside this machinery: its drop-in constraint requires consuming the production ranker’s existing feature space, so CCFormer tokenizes raw user, behavior, and target feature fields directly instead of semantic IDs (§3).

User/context encoding. The generative-recommendation models adopt a common encoder–decoder interface: a Transformer-based encoder maps (u, S^u, x^u) to a contextualized memory C that serves as the key/value context for cross-attention in the decoders. TGR-Reason reuses TGR-GenRec’s encoder–decoder backbone directly (§5.2.3), so reasoning and memory signals enter generation without any representation conversion.

Training and serving infrastructure. TGR is trained on *Numerous-Torch* (§3.3), Tencent’s Megatron-style 3D-parallel trainer with cross-GPU embedding caching. Default deployments run on clusters of NVIDIA H20 nodes with NVLink intra-node and 400 Gbps RDMA inter-node. We use BFloat16 mixed precision, ZeRO-1 sharding for dense parameters, gradient checkpointing on the deepest layers, and TensorRT-style fused kernels for the generative decoder. Online serving runs on NVIDIA L20 GPUs with KV-caching enabled by default.

3. TGR-GenRank: Generative-Paradigm Ranking

TGR-GenRank is the TGR Stack’s generative-paradigm ranking direction: it keeps the cascade in place but replaces the production ranker itself with a Transformer that borrows the generative paradigm—unified tokenized inputs, an LLM-style backbone with predictable scaling behavior, and LLM-style training infrastructure—while still emitting per-item multi-task

scores. Its production model is **CCFormer** [22], an efficient Transformer backbone that unifies *cross-field feature interaction* and *compressed long-sequence modeling*. CCFormer confronts the central deployment obstacle of the HSTU / OneTrans line of work [6, 8]: attention-based rankers unlock scaling laws, but quadratic self-attention over long behavior sequences is unaffordable under industrial latency and resource budgets. By resolving this tension architecturally rather than by truncating or pre-compressing the sequence, CCFormer is simultaneously *more accurate and cheaper* than its attention-based predecessors: it trains $2.21\times$ faster than a strong HSTU baseline at higher AUC and GAUC. These offline gains translate into consistent online gains: CCFormer has been validated through online A/B experiments across *five* recommendation ranking scenarios at Tencent, delivering statistically significant lifts on the core business metrics in each scenario.

Consistent with the industrial focus of this report, we keep the architectural description compact (§3.2) and devote most of the section to the training and serving recipe (§3.3) and the experimental evidence—offline comparisons, scaling behavior, ablations, and online A/B results (§3.4–§3.6).

3.1. Motivation

Industrial rankers differ primarily in *how they realize feature interaction*: over which tokens, at what asymptotic cost, and how much of the user history is retained. Under this view, ranking architectures deployed at scale fall into three families, each with a characteristic limitation.

Conventional DLRMs. The incumbent paradigm embeds sparse categorical features, concatenates them with dense features, and applies hand-designed crossing modules: explicit low-order crossing with deep networks [3], target-aware attention pooling [26], or memory-augmented compression of long histories [27]. Such models are cheap to serve but treat behaviors as aggregated or short-range features rather than as sequential evidence, so pooled representations discard fine-grained signals and capacity is spent on static feature pairs instead of history–target interaction. They also saturate with added capacity, offering no usable scaling law.

Attention-based sequential rankers. The second family models behaviors token by token with self-attention [4], and its industrial variants jointly attend over behavior sequences, item features, and user profiles in a single scalable backbone [6, 8, 28]. These models adaptively identify informative historical interactions and exhibit predictable scaling laws in *both* sequence length and model capacity—the only family that scales along both axes. Their obstacle is cost: self-attention is quadratic in sequence length in both computation and memory, which is prohibitive when scoring massive traffic under latency budgets of tens of milliseconds. Two lossy remedies are common in production. *Compress-then-interact* methods condense the sequence before feature interaction, sacrificing fine-grained signals and weakening history–target interaction. *Retrieve-or-truncate* methods (the SIM / TWIN lineage [29, 30]) retain token-level modeling only within a target-relevant subsequence, discarding long-term interests while still paying full attention inside the retained window. Efficient-attention variants, in turn, optimize mainly the sequence-modeling term and remain weak at the cross-field interaction on which ranking depends.

Token-mixing rankers. A third family sequentializes the feature space into unified tokens and exchanges information across them with lightweight mixing operators derived from MLP-based alternatives to Vision Transformers, avoiding quadratic self-attention. RankMixer [31] combines multi-head token mixing with per-token FFNs to model heterogeneous feature subspaces at near-linear cost, recovering predictable *capacity* scaling on hardware-efficient primi-

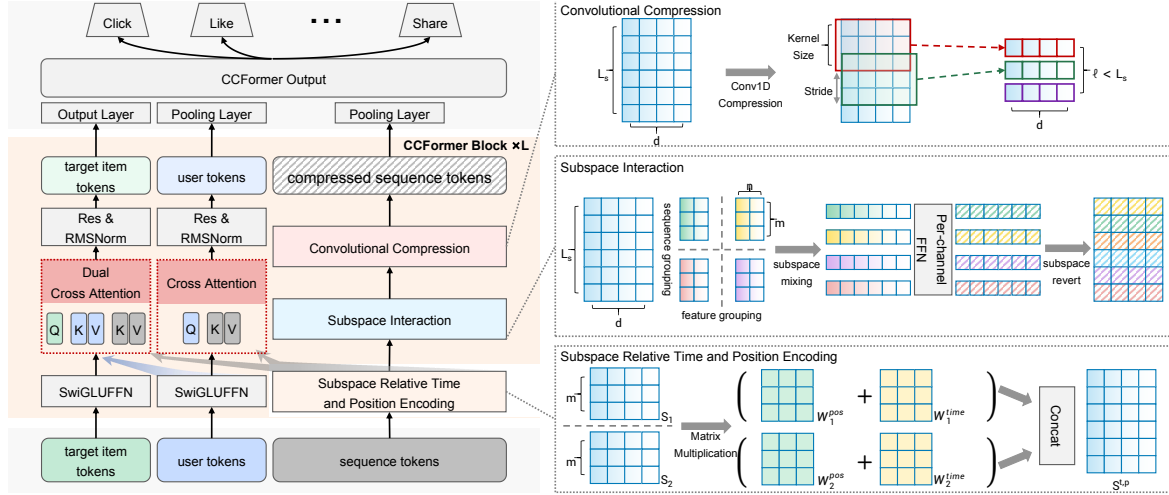


Figure 2 | Overview of CCFormer. The input feature space is partitioned into user, behavior-sequence, and target-item fields; each block combines feature-field separated cross attention, subspace token mixing with relative temporal-positional encoding, and convolutional sequence compression, so that global self-attention over the concatenated token set is never computed.

tives. Their limitation is scope: these operators target a small set of unified feature tokens, not behavior sequences of hundreds to thousands of items. Long-term interest must still be summarized into few tokens before mixing, reintroducing the information loss of the first family.

Our approach. The three families point to one unmet requirement: *sufficient and efficient feature interaction over full-length behavior sequences*. Insufficient sequence-level crossing weakens fine-grained preference discovery, whereas exhaustive self-attention provides the crossing at a cost that is deployable only by discarding history. CCFormer therefore decouples feature interaction into two orthogonal components and makes each subquadratic by construction: *cross-field* interaction among user, sequence, and target fields, realized by directed field-separated cross attention that computes only the interaction directions ranking requires; and *intra-sequence* interaction among behavior tokens, realized by extending token mixing from short unified token sets to the full sequence, combined with hierarchical compression that gives deeper blocks whole-history coverage at geometrically decreasing cost. CCFormer thus inherits the scaling behavior of the attention family and the efficiency of the token-mixing family without retrieving or truncating the sequence in advance.

3.2. CCFormer Architecture in Brief

CCFormer partitions the input feature space into three semantic fields—user-profile tokens $\mathbf{U} \in \mathbb{R}^{L_u \times d}$, behavior-sequence tokens $\mathbf{S} \in \mathbb{R}^{L_s \times d}$ (one fine-grained token per historical item), and target-item tokens $\mathbf{T} \in \mathbb{R}^{L_t \times d}$ —and stacks L interaction blocks that iteratively refine all three (Figure 2). After the final block, the normalized field representations are aggregated and fed to task-specific heads for multi-task prediction. Four mechanisms, applied per block, replace global self-attention:

1. **Feature-field separated cross attention.** Instead of undirected attention over all concatenated tokens (which treats semantically different fields uniformly and wastes computation), three *directed* attention flows implement the heterogeneous interactions of ranking: a user→sequence flow (preference retrieval over the full history), a target→sequence flow (target–history relevance matching), and a target→user flow (target–user compatibility).

RMSNorm aligns the scales of the fields and lightweight SwiGLU FFNs refine user and target tokens before crossing. Crucially, target tokens act only as queries and never attend to one another—a design choice that later enables single-pass scoring of all candidates (§3.3).

2. **Relative temporal-positional encoding in token subspaces.** Recency-aware order information is injected *within local groups of m behaviors*: a learnable time-decay weight $\alpha \beta^{|t_i - t_j|^\gamma}$ (with $\beta \in (0, 1)$, so temporally proximate behaviors receive larger weights) plus a learnable relative positional bias, at $\mathcal{O}(L_s m)$ cost rather than the $\mathcal{O}(L_s^2)$ of attention-matrix biases.
3. **Long-sequence subspace token mixing.** The sequence tensor is reshaped into compact subspaces, each jointly spanning m adjacent behavior tokens and n hidden channels, and each channel group is processed by an independent gated per-channel feed-forward network. This forces token-level and channel-level signals to interact directly within localized subspaces, at cost linear in L_s . To our knowledge this is the first attempt to extend the token-mixing paradigm of industrial ranking [31] from small sets of unified feature tokens to full-length behavior sequences.
4. **Hierarchical sequence compression.** After each block, a strided one-dimensional convolution (kernel k , stride s) fuses adjacent behavior tokens and hands a shortened sequence to the next block. The receptive field of each surviving token thus expands progressively with depth: shallow layers capture short-term, fine-grained patterns over the raw sequence, while deep layers extract abstract, long-term preference signals over the full history. Because later blocks operate on geometrically shorter sequences, total computation drops substantially while access to the entire sequence is preserved—compression *across* layers complements mixing *within* layers.

3.3. Training and Serving at Industrial Scale

CCFormer’s deployment recipe is as much a part of TGR-GenRank as its architecture; four ingredients carry the model from research code to main-traffic serving.

Numerous-Torch infrastructure. CCFormer is built on top of *Numerous-Torch*, Tencent’s production-grade training and inference infrastructure for large-scale recommendation models, whose role is to bridge flexible model innovation and industrial-scale deployment. On the modeling side it keeps a PyTorch-native development experience, so emerging advances from the rapidly evolving LLM community can be reused directly; on the system side it supplies distributed data ingestion, large-scale sparse-feature processing, sparse-dense joint training, checkpoint recovery, model export, and continual training on fresh user-feedback logs.

Mixed-precision training. The main model computation runs in BF16/FP16 mixed precision, reducing memory overhead by more than 35%—headroom that is spent on larger batches or larger models under the same training budget.

Sparse-parameter compression. Sparse user- and item-ID embeddings dominate the parameter count of industrial rankers. CCFormer stores ID-feature embedding tables with INT8 symmetric linear quantization ($\sim 70\%$ compression over full-precision storage) and shrinks the tables themselves with a double-hashing strategy (a further $\sim 50\%$ reduction in sparse parameter size). Together these substantially cut memory footprint and communication cost during training without degrading training effectiveness.

Table 2 | Offline comparison on public benchmarks (mean $\pm$ std over 5 runs). Δ AUC is the relative improvement over the DIN base model. Best in **bold**, runner-up underlined.

Method	Taobao		KuaiRec	
	AUC (%)	Δ AUC (%)	AUC (%)	Δ AUC (%)
DIN	88.33 $\pm$ 0.22	0.00	80.22 $\pm$ 0.58	0.00
DeepFM	89.06 $\pm$ 0.42	1.90	80.14 $\pm$ 0.51	-0.26
SASRec	88.52 $\pm$ 0.65	0.50	79.83 $\pm$ 0.70	-1.29
MIMN	91.79 $\pm$ 0.32	9.03	81.79 $\pm$ 0.81	5.20
HSTU	91.40 $\pm$ 0.25	8.01	82.62 $\pm$ 0.39	7.94
OneTrans	91.35 $\pm$ 0.61	7.88	<u>82.76$\pm$0.59</u>	<u>8.41</u>
STCA	<u>92.81$\pm$0.50</u>	<u>11.69</u>	82.18 $\pm$ 0.16	6.49
CCFormer	93.67$\pm$0.32	13.93	83.35$\pm$0.29	10.36

Table 3 | Offline comparison on the industrial dataset (4B+ samples, sequence length 1000). Δ columns are relative improvements over the HSTU base model. Best in **bold**, runner-up underlined.

Model	AUC (%)	Δ AUC (%)	GAUC (%)	Δ GAUC (%)
HSTU	77.66	0.00	70.86	0.00
OneTrans	77.69	0.11	70.90	0.19
STCA	<u>77.73</u>	<u>0.25</u>	<u>70.95</u>	<u>0.43</u>
CCFormer	77.94	1.01	71.36	2.40

Single-pass parallel scoring of candidates. Because target tokens never attend to each other, there is no cross-target information leakage, and all candidate items of a request can be packed into the target field and forwarded *once*: user-field and sequence-field computation is shared across candidates, and the model emits multi-task scores for every candidate in a single pass. Relative to the pointwise per-candidate inference of the incumbent DLRM, this raises online peak QPS by 30% at unchanged inference resources—even though CCFormer’s per-sample computational cost is 20 $\times$ higher.

Default configuration. Industrial experiments use feature-token dimensionality $d = 256$, 8 CCFormer blocks, subspace group sizes $m = 8$ and $n = 16$, compression kernel $k = 3$ with stride $s = 2$, Adam with learning rate 10^{-4} , and batch size 4096; all offline comparisons below run on an identical hardware/software stack of 16 NVIDIA H20 GPUs. Per-model default configurations across the stack are consolidated in Appendix C.

3.4. Offline Experiments

Setup. We evaluate on two public benchmarks—Taobao¹ and KuaiRec [32], with behavior sequences truncated at length 200—and an industrial dataset of one month of production logs from a Tencent recommendation system: over 4 billion samples, 30 million users, and 10 million items, with behavior sequence length 1000. Public-benchmark baselines span the conventional DLRM paradigm (DIN [26], DeepFM [3], SASRec [4], MIMN [27]) and attention-based sequential rankers (HSTU [6], OneTrans [8], STCA [28]); the industrial comparison uses the production-oriented subset (HSTU, OneTrans, STCA) with both AUC and GAUC. Δ AUC / Δ GAUC denote relative improvements in the RelAImpr sense, and every method receives its own TPE-based hyperparameter search for fairness.

¹<https://tianchi.aliyun.com/dataset/649>

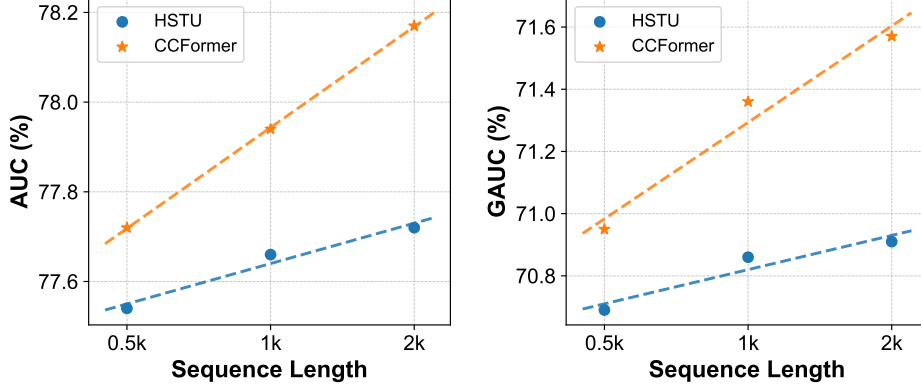


Figure 3 | Sequence-length scaling on the industrial dataset (lengths 500, 1000, 2000). CCFormer improves predictably with longer sequences and dominates HSTU at every length; with 0.5k behavior tokens it already matches the AUC and surpasses the GAUC of HSTU consuming 2k tokens.

Overall results. CCFormer achieves the best score on every dataset and metric (Tables 2 and 3). On Taobao and KuaiRec it reaches 93.67% and 83.35% AUC, beating the strongest sequential baseline on each dataset (STCA and OneTrans, respectively) by 0.86 and 0.59 points, and it maintains a consistent margin over HSTU, OneTrans, and STCA when the KuaiRec sequence length is varied over {200, 500, 1000}. On the industrial dataset it improves over HSTU by +0.28 AUC and +0.50 GAUC points (1.01% and 2.40% relative)—a large margin at production scale, where 0.1%-level AUC moves are considered shippable—and still clears the strongest baseline STCA by +0.21 AUC and +0.41 GAUC points.

Hyperparameter robustness. The two long-sequence-specific hyperparameter families are insensitive over wide ranges: varying the compression kernel k from 2 to 7 moves AUC only within 77.92–77.95%, and all tested (m, n) subspace group sizes outperform HSTU with limited variation. In practice, CCFormer transfers to new scenarios without delicate tuning.

3.5. Scaling Behavior

The premise of generative-paradigm ranking is that ranking quality can be bought by FLOPs; the question is at what exchange rate. We examine the two axes that matter most for sequential rankers—sequence length and model capacity—holding the hardware/software stack fixed and reporting GFLOPs alongside quality.

Sequence-length scaling. As the industrial sequence length grows from 0.5k to 2k, CCFormer improves monotonically from 77.72% to 78.17% AUC and from 70.95% to 71.57% GAUC, with average relative gains over HSTU of 1.08% AUC and 2.37% GAUC across length settings—and the margin *widens* as sequences lengthen (Figure 3). Most strikingly, **CCFormer with 0.5k behavior tokens already matches the AUC and surpasses the GAUC of HSTU with 2k tokens**: subspace token mixing extracts expressive patterns from the full history, while hierarchical compression preserves a large receptive field at a fraction of the cost.

Model-size scaling. Varying the feature dimensionality over {128, 256, 512} (Table 4), CCFormer again scales predictably (77.65% $\rightarrow$ 78.13% AUC, 70.76% $\rightarrow$ 71.52% GAUC, average relative gains of 1.02% AUC and 1.93% GAUC over HSTU at matched capacity) while consuming roughly half the GFLOPs of HSTU at every capacity point—at $d = 128$ it comes within

Table 4 | Model-size scaling on the industrial dataset. Δ columns are relative improvements over HSTU at the same feature dimensionality.

Model	Dim.	AUC (%)	Δ AUC (%)	GAUC (%)	Δ GAUC (%)	GFLOPs/sample
HSTU	128	77.42	0.00	70.55	0.00	28.87
CCFormer	128	77.65	0.84	70.76	1.02	9.34
HSTU	256	77.66	0.00	70.86	0.00	38.37
CCFormer	256	77.94	1.01	71.36	2.40	18.28
HSTU	512	77.79	0.00	71.02	0.00	63.47
CCFormer	512	78.13	1.22	71.52	2.38	36.14

Table 5 | Ablation study on the industrial dataset. Training speedup is the training throughput relative to HSTU; Δ columns are relative improvements over HSTU. Best in **bold**, runner-up underlined.

Variant	Speedup	AUC (%)	Δ AUC (%)	GAUC (%)	Δ GAUC (%)
HSTU (base)	1.00 $\times$	77.66	0.00	70.86	0.00
CCFormer (full)	2.21 $\times$	<u>77.94</u>	<u>1.01</u>	71.36	2.40
w/o relative temporal-pos. encoding	<u>2.35</u> $\times$	77.85	0.69	71.13	1.29
token mixing $\rightarrow$ self-attn	1.41 $\times$	77.96	1.08	71.21	1.68
w/o long-seq. token mixing	2.40 $\times$	77.73	0.25	71.02	0.77
w/o sequence compression	1.29 $\times$	77.96	1.08	<u>71.25</u>	<u>1.87</u>

0.01 AUC points of HSTU at $d = 256$ using $4\times$ fewer GFLOPs per sample. The two analyses together show that CCFormer offers a strictly better effectiveness–efficiency exchange rate, which is what makes scaling *deployable* rather than merely demonstrable.

Ablations: where the speed and the quality come from. Table 5 disentangles the contributions of the three core designs, measuring training speedup as relative training throughput (samples per unit time) against HSTU under an identical stack. Three findings stand out. First, *sequence compression is the efficiency lever*: removing it leaves AUC essentially unchanged (77.96%) but collapses the training speedup from $2.21\times$ to $1.29\times$. Second, *subspace token mixing is the quality workhorse*: removing it causes the largest degradation (-0.21 AUC / -0.34 GAUC points), while replacing it with standard self-attention buys only $+0.02$ AUC points, *loses* 0.15 GAUC points, and drops the speedup to $1.41\times$ —costly global attention over the sequence is unnecessary. Third, relative temporal-positional encoding contributes $+0.09$ AUC / $+0.23$ GAUC points by separating recent behaviors from outdated ones.

3.6. Online Deployment and A/B Results

CCFormer has been A/B-tested in five Tencent scenarios covering video recommendation, content feeds, and advertising ranking. The incumbents differ by scenario, which makes the comparison informative in two directions: Scenario 1 replaces a long-standing, iteratively optimized DLRM ranker, whereas Scenarios 2–5 replace an already deployed HSTU model. Each experimental group served more than one million exposed users per day over a two-week window. Every lift reported is statistically significant (two-sample t -test, $p < 0.05$), confirming the reliability of the improvements in the production environment.

In Scenario 1 (video recommendation), where CCFormer replaces the long-standing production DLRM, the gains span the full engagement funnel—click (CTR $+3.57\%$, UCTR $+1.93\%$), consumption (Page View $+3.86\%$, Video View $+3.47\%$, Watch Time $+1.29\%$), audience (Unique Viewer $+2.61\%$), and short-term retention (3-day Activeness $+1.93\%$)—and both Scenario 1 and

Table 6 | Online A/B gains of CCFormer across five Tencent production scenarios. Each experimental group served >1M exposed users per day over a two-week window; all lifts are statistically significant (two-sample t -test, $p < 0.05$).

Scenario 1								
Metric	CTR	UCTR	3-day Activeness	Video View	Unique Viewer	Page View	Watch Time	Ad revenue
Lift (%)	3.57	1.93	1.93	3.47	2.61	3.86	1.29	1.64
Scenario 2								
Metric	Ad view				Ad revenue			
Lift (%)	1.43				1.71			
Scenario 3								
Metric	CTR		UCTR		New User Page View		Ad revenue	
Lift (%)	2.02		1.67		4.01		1.29	
Scenario 4								
Metric	Page View		Finish		Watch Time		Follow	
Lift (%)	0.55		0.98		0.42		2.18	
Scenario 5								
Metric	Page View		Watch Time		Share		CTR	
Lift (%)	0.45		1.26		3.49		1.01	

the advertising-ranking Scenario 2 lift advertising revenue (+1.64% and +1.71%), the key monetization metric. Following the A/B tests, CCFormer was fully launched in Scenarios 1 and 2 and now serves the entire production traffic of both, with the gains remaining stable after full deployment.

4. TGR-GenRec: From Next-Token to Next-Slate Generation

This section presents **TGR-GenRec**, the end-to-end generation direction of the TGR Stack and its most fully developed component. TGR-GenRec is a *direction*, not a single model: it spans the two generation paradigms of generative recommendation, and currently ships one production model for each—**BARGE** [23] under the *next-token-prediction* (NTP) paradigm that generates one item at a time, and **HiGR** [24] under the *slate-generation* paradigm that emits the whole result list as one structured object. We first position the two paradigms and models (§4.1); §4.2 and §4.3 then present the two works independently, as parallel and self-contained units, each with its own motivation, method, offline results, and deployment.

4.1. Two Generation Paradigms: Next-Token and Next-Slate Prediction

The long-term goal of TGR-GenRec is to replace the cascade’s learned decision stages with generative models that directly produce the ordered recommendation result. The deployments reported here are intermediate realizations of that goal rather than a uniform full-pipeline replacement. In production this goal is pursued through two generation paradigms that pose logically independent problems, attacked by two separate works. We present them in order of generation granularity—from single-item NTP to whole-slate generation:

- **The NTP paradigm — generating one item faithfully (BARGE, §4.2).** The mainstream generative formulation (the TIGER lineage) tokenizes each item into a hierarchical SID tuple and predicts the next item token by token. But this formulation was borrowed from natural language, where every token carries stand-alone semantics; a hierarchical SID codeword is meaningful only jointly with its prefix. The mismatch creates two structural gaps in *any* NTP-style SID generator: flattening dissolves item boundaries at the encoder, and hierarchical decoding accumulates semantic drift—an error at any layer strands the search in a wrong subtree. BARGE’s answer is to adapt the NTP mechanism itself to the recommendation task: restore item structure at the encoder (ICA), rerank decoding paths for global coherence (HPR), and decode through two orthogonal quantization channels (DPD).
- **The slate-generation (next-slate-prediction, NSP) paradigm — generating the whole list (HiGR, §4.3).** Even a perfectly faithful next-item generator decides items one at a time, while our surfaces present an ordered slate in a single display: quality lives at the list level—ordering, compatibility, diversity—and decoding $M \times D$ tokens item by item is too slow under real-time constraints, with token-level likelihood optimizing neither. HiGR’s answer is to make the *list* the unit of generation: hierarchical semantic IDs with controllable prefixes (PCRQ-VAE), coarse-to-fine slate planning and item decoding (HSD), and listwise multi-objective alignment (ORPO).

The two models are orthogonal by construction—BARGE hardens the *generation mechanism* within the NTP paradigm, HiGR changes the *output structure and objectives* of generation—and their fixes act on disjoint failure modes. Table 7 summarizes the positioning; both were shipped under the same production discipline (principle P1, §2.1): drop-in deployment behind the existing filter stack at matched or better latency and GPU budgets.

Table 7 | The two TGR-GenRec production models, one per generation paradigm. BARGE and HiGR arise from different motivations and repair complementary failure modes of end-to-end generative recommendation.

	BARGE [23]	HiGR [24]
Generation paradigm	Next-token prediction (single item)	Slate generation (whole list)
Motivating gap	NLP-born autoregression structurally mismatches SID-based recommendation	Item-level generation misaligns with list-level UX and latency
Granularity attacked	Token/path level (<i>how</i> to generate)	Slate level (<i>what</i> to generate)
Failure modes repaired	Item-boundary loss; hierarchical semantic drift	Entangled SID prefixes; $M \times D$ -token decoding cost; tokenwise objectives
Tokenizer	OSQ-VAE (orthogonal dual-channel)	PCRQ-VAE (prefix-contrastive)
Decoder	Dual-decoder + HPR reranking, OR-fusion	HSD: slate planner + item generator
Training / alignment	NTP + symmetric InfoNCE (label-free)	NTP + ORPO listwise (3 objectives)
Serving-cost delta	Zero (params, beam, latency unchanged)	−60% GPU; P99 < 50 ms
Online A/B	+0.60% CTR, +1.34% unique clicking users, +1.70% reading time (6% traffic)	+1.22% watch time, +1.73% views (5% traffic)

4.2. BARGE: Adapting Next-Token Prediction for Industrial Recommendation

BARGE (Bridging **A**uto**R**egressive **G**eneration for **r**ecommendation) [23] is the TGR-GenRec production model under the *next-token-prediction* paradigm (§4.1): each item is represented as a hierarchical SID tuple and the next item is generated token by token, in the mainstream TIGER-style formulation. Rather than replacing this paradigm, BARGE hardens *how* SID sequences are generated, closing two structural gaps between the NLP-born autoregressive formulation and the recommendation task. BARGE was designed under a hard production constraint—*no increase in parameters, beam budget, or serving latency*—and has been deployed on Tencent’s commercial media platform, where a 6% live-traffic A/B test delivered +0.60% click-through rate, +1.34% unique clicking users, and +1.70% total reading time over the incumbent multi-stage system.

4.2.1. Motivation: Two Structural Gaps at Industrial Scale

Generative recommendation avoids exhaustive scoring over the entire item corpus by representing each item as a sequence of hierarchical semantic IDs (SIDs) and autoregressively generating the target sequence. Although this formulation enables efficient token-level generation, it introduces a mismatch between the modeling unit and the task unit: the backbone operates on individual SID tokens, whereas user interactions and recommendation targets are defined at the item level. In RQ-VAE-based implementations, these SIDs form an ordered, coarse-to-fine code path. Flattening these paths removes explicit item grouping from the encoder, while autoregressive decoding exposes the hierarchical path to cumulative prefix errors. These properties give rise to two structural gaps in the encode–decode pipeline, which become particularly consequential at industrial scale.

(P1) Item-boundary gap. Existing RQ-VAE-based generative recommenders [33, 34] tokenize each item into an L -level SID and concatenate all SIDs in the user history into a single token sequence. Although this representation is convenient for standard sequence models, it removes explicit item boundaries before the sequence enters the encoder. Self-attention therefore processes all SID tokens in the same manner and must infer which tokens belong to the same item mainly from positional and contextual signals.

A more subtle ambiguity arises from the hierarchical nature of RQ-VAE codewords. The semantics of a codeword depend on its prefix, whereas its input embedding is typically deter-

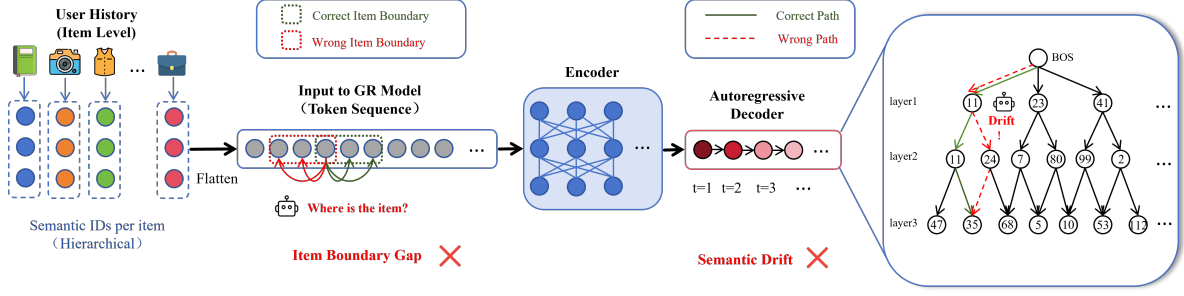


Figure 4 | Two structural gaps in generative recommendation. **Left (Item-Boundary Gap, P1):** after each item is tokenized into L hierarchical semantic IDs and flattened into a single sequence, item boundaries vanish from the input to the GR model, and the encoder is no longer explicitly informed which tokens belong to the same item (“Where is the item?”). **Right (Semantic Drift, P2):** on the hierarchical codebook tree, an error at any layer redirects decoding into a wrong subtree, so that the target leaf becomes unreachable along the selected path no matter how the remaining tokens are scored.

mined only by its code index and quantization level. Consider two items with SIDs $\langle 1, 3, 5 \rangle$ and $\langle 7, 3, 8 \rangle$. At the second level, both items use codeword 3 and therefore receive the same initial lookup embedding for that token. However, the partial paths $\langle 1, 3 \rangle$ and $\langle 7, 3 \rangle$ correspond to different nodes in the hierarchical codebook and need not represent the same item-level semantics. The downstream encoder must consequently recover this prefix-conditioned distinction through contextualization.

After flattening, the encoder must simultaneously reconstruct item membership and disambiguate the prefix-dependent semantics of individual codewords. This places the burden of item-level preference modeling on implicit token interactions, even though recommendation is fundamentally performed at the item level. As illustrated on the left of Figure 4, this mismatch motivates an explicit item-level aggregation mechanism that preserves token-level information while restoring the structural context shared by the SID tokens of each item.

(P2) Semantic drift. Hierarchical SIDs define a tree-structured decoding path, in which each predicted codeword constrains the subtrees reachable at subsequent layers. An early error therefore redirects decoding to an incorrect subtree and makes the target item unreachable along the selected path. Although beam search retains multiple partial paths, it ranks them mainly by accumulated token log-probabilities without explicitly evaluating their global semantic coherence. We refer to this prefix-induced error propagation as *semantic drift*, as illustrated on the right of Figure 4.

Table 8 quantifies this effect on the TIGER baseline. At the deepest layer c_3 , the probability assigned to the target codeword decreases from 0.787 under teacher forcing to 0.015 under autoregressive decoding, a roughly $52\times$ drop. Under autoregressive decoding, the c_3 accuracy is 77.0% with a correct prefix but only 0.6% with an erroneous one, yielding a gap of more than $128\times$. This result shows that deep-layer failures arise largely from preceding path errors, rather than solely from the prediction difficulty of the current layer.

Crucially for production, the brute-force remedy—enlarging the beam width B —is not viable: inference latency scales approximately linearly with B , while beam-dependent decoding memory grows with $B \times L$. Moreover, a larger beam only preserves more candidate paths without explicitly correcting the underlying semantic drift. This trade-off is particularly acute at platform scale, where the item corpus can contain tens or even hundreds of millions of items and follows a highly skewed long-tail distribution, making path-consistent decoding more important under strict online serving constraints.

Together, these observations motivate BARGE. Item Context-Aware Attention (ICA) re-

Table 8 | Per-layer prediction quality under Teacher Forcing (TF) and Autoregressive (AR) decoding on Amazon Beauty (TIGER baseline, greedy decoding). The last two rows split AR results at c_3 by prefix correctness, exposing the error cascade that motivates BARGE.

Layer / Mode	Mismatch ↓	Rank ↓	Prob ↑
c_1 (TF / AR)	0.915	66.9	0.046
c_2 (TF)	0.864	25.3	0.102
c_2 (AR)	0.981	108.2	0.015
c_3 (TF)	0.195	7.4	0.787
c_3 (AR)	0.984	119.5	0.015
c_3 (AR, prefix correct)	0.230	3.7	0.738
c_3 (AR, prefix error)	0.994	121.1	0.006

stores item-level structure in the encoder, Hierarchical Path Reranking (HPR) evaluates global path coherence within each decoding channel, and Dual-Path Decoding (DPD) provides a structurally complementary recovery path. BARGE thereby addresses the two structural gaps without requiring a larger model or a wider beam.

4.2.2. Method Overview

As shown in Figure 5, BARGE addresses the two structural gaps through three lightweight and complementary modules:

- **Item Context-Aware Attention (ICA)** addresses the item-boundary gap on the encoder side. For each item, ICA uses a learnable query to aggregate its L SID-token embeddings through cross-attention, and injects the resulting item-level context into every token through a gated residual connection. When the gate approaches zero, the module recovers an identity path and preserves the original token representation. Empirically, the average gate activations across the four SID levels concentrate around 0.35–0.38 on the two public datasets, indicating that ICA learns a moderate level of context injection rather than overwriting the token-level signal.
- **Hierarchical Path Reranking (HPR)** addresses semantic drift within each decoding channel. At layer l , a dual-tower scorer evaluates the compatibility between the decoder’s pre-generation hidden state $\mathbf{h}_0$ and the cumulative path embedding $\mathbf{p}^{(l)} = \sum_{j=1}^l \mathbf{e}_{c_j}$. HPR is jointly trained with the main model using symmetric InfoNCE: each positive pair matches $\mathbf{h}_0$ with its ground-truth cumulative path, while the negatives combine in-batch paths, high-probability non-ground-truth prefixes sampled from the next-token prediction distribution, and optional business-feedback negatives such as impressed-but-unclicked items. These prefix-aware negatives expose HPR to plausible drift patterns during training. At inference, HPR combines its path-compatibility score with the generation log-probability at weight λ to rerank a Top- N pool, while retaining only the Top- B paths for the next layer. The outgoing beam width therefore remains unchanged, and the context-side representation is shared across all candidate paths to limit the additional scoring cost.
- **Dual-Path Decoding (DPD)** addresses semantic drift from a complementary decoding direction. Its Orthogonal Split-and-Quantize VAE (OSQ-VAE) applies a learnable Householder-parameterized rotation R , with $R^\top R = I$ by construction, and splits the rotated item representation into two orthogonal coordinate subspaces. The two subspaces are quantized by independent residual codebook stacks, producing two SID channels for each item. A shared encoder feeds two channel-specific decoder towers, each equipped with private output heads and HPR scorers. Their item-space scores are merged through LSE-based

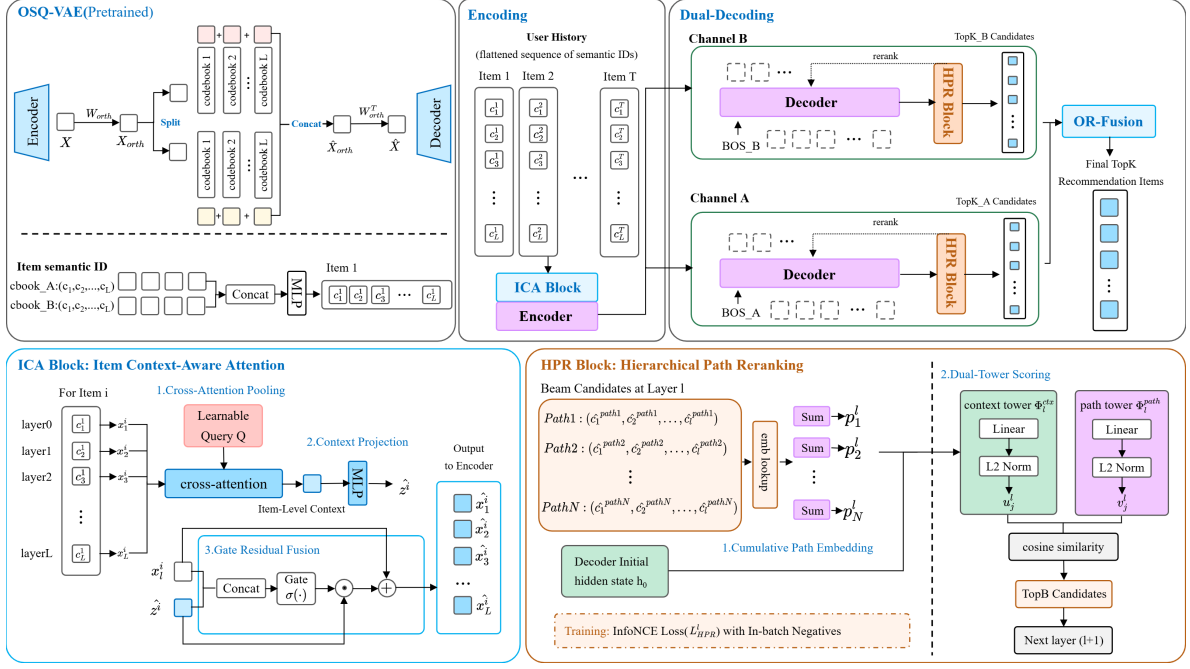


Figure 5 | Overview of BARGE. The OSQ-VAE tokenizes each item into two channel-specific SID tuples via orthogonal rotation and dual codebook stacks; the user history passes through the ICA block before the shared Transformer encoder; two decoders, each followed by an HPR block, run in parallel and their Top-K lists are merged by OR-fusion.

soft-OR fusion, allowing an item ranked highly by either channel to be recovered. DPD keeps the per-channel beam width and final Top-K output length unchanged, although the two decoding branches introduce additional decoder computation relative to a single-channel model.

The two decoding channels exhibit strong empirical complementarity. At $K = 10$, their candidate pools have Jaccard similarities of only 0.183 on Beauty and 0.172 on Sports. Among the test cases successfully recalled by OR fusion, 15.6% and 24.3% are contributed exclusively by one channel on the two datasets, respectively. Figure 6 further shows that the cumulative hit-rate advantage of ICA over its no-ICA counterpart increases at deeper SID levels, a trend consistent with improved resistance to hierarchical error propagation.

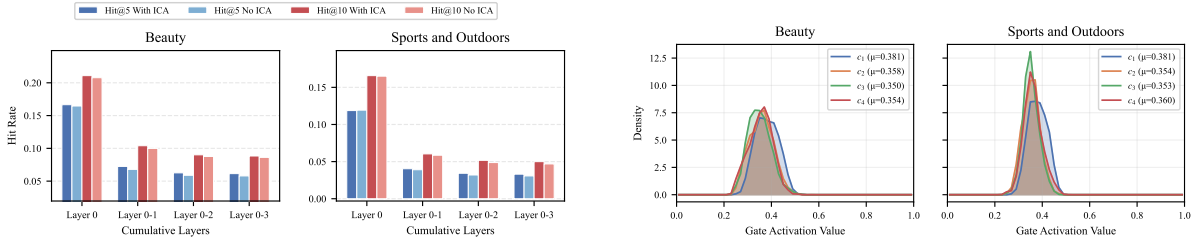


Figure 6 | *Left*: cumulative hit rate with and without ICA across SID layers—the gap widens at deeper layers, showing stronger resistance to semantic drift. *Right*: gate activation distribution across layers, stably concentrated around 0.35–0.38, confirming selective and consistent injection of item-level context.

Table 9 | BARGE vs. the strongest published baselines on Amazon Beauty and Sports (R@10 / N@10). BARGE-base replaces the 4-layer learned codebook with TIGER’s 3-layer codebook plus a collision-resolving ID.

Method	Beauty		Sports	
	R@10	N@10	R@10	N@10
TIGER [33]	0.0648	0.0384	0.0400	0.0225
HSTU [6]	0.0704	0.0389	0.0414	0.0215
COBRA [35]	0.0725	0.0456	0.0434	0.0257
ActionPiece [36]	0.0775	0.0424	0.0500	0.0264
APAO-pointwise [37]	0.0795	0.0453	0.0444	0.0237
BARGE-base	0.0896	0.0515	0.0513	0.0285
BARGE	0.0927	0.0547	0.0544	0.0308

4.2.3. Offline Results

Public benchmark results. On Amazon Beauty and Sports (leave-one-out, full-catalog evaluation), BARGE achieves the best score on every metric against both discriminative and generative baselines (Table 9); e.g., on Beauty it lifts R@10 from 0.0775 (ActionPiece) to 0.0927 (+19.6%) and beats TIGER by 43.0%. A codebook-controlled variant (BARGE-base) already surpasses all prior baselines, isolating the gains of the three structural modules from the codebook design. Component ablations show every module helps independently and their gains are largely additive, consistent with the modules acting on disjoint failure sources.

Industrial-scale offline evaluation. We evaluate BARGE in two large-scale recommendation scenarios on Tencent’s commercial media platform. The two scenarios differ in both corpus scale and incumbent architecture, allowing us to examine BARGE under distinct production settings. Scenario 1 uses an 11-day observation window, with the first 10 days used for training and the final day for evaluation. The resulting dataset covers millions of users, hundreds of millions of interactions, and hundreds of thousands of items. Scenario 2 operates at a substantially larger scale, covering tens of millions of users, tens of millions of candidate items, and tens of billions of user-item interactions.

The deployed production baselines also differ by scenario. Scenario 1 compares BARGE with a GraphSAGE-based retrieval model and Neural Approximate Nearest Neighbor Search (NANN), while Scenario 2 uses a GraphSAGE-based retrieval model and a BERT-based sequential recommender. These models are production baselines currently deployed in their respective scenarios. In addition, we compare BARGE with OneRec as a generative baseline in both scenarios. Because the source code and model checkpoints corresponding to the production system described in the original OneRec technical report [17] are not publicly available, we implement OneRec in-house following its published architecture and methodological description. This in-house OneRec baseline is used only for offline comparison and is not deployed in either scenario.

As shown in Table 10, BARGE consistently achieves the best performance across all four Hit@K metrics in both scenarios. In Scenario 1, BARGE improves Hit@5 by 10.2% and Hit@10 by 6.2% relative to the strongest baseline, OneRec. The gain is even more pronounced in Scenario 2, where BARGE improves Hit@5 by 16.9% and Hit@10 by 14.7% over OneRec. Its relative improvement remains 12.6% at Hit@50, showing that the gain extends beyond the first few positions of the ranked list.

The compared systems span graph-based retrieval, Neural Approximate Nearest Neighbor Search, BERT-based sequential recommendation, and semantic-ID-based generative recommendation. GraphSAGE-based, NANN, and BERT-based models operate primarily on atomic item representations and do not explicitly model the coarse-grained semantic structure shared by re-

Table 10 | Offline performance in two large-scale recommendation scenarios on Tencent’s commercial media platform. Underlined and bold values denote the best baseline and the best overall result, respectively.

Method	Hit@5	Hit@10	Hit@20	Hit@50
<i>Scenario 1</i>				
GraphSAGE-based	0.2932	0.3743	0.4650	0.5951
NANN	0.4416	0.4946	0.5636	0.6760
OneRec	<u>0.5459</u>	<u>0.6132</u>	<u>0.6729</u>	<u>0.7348</u>
BARGE	0.6015	0.6510	0.6967	0.7520
<i>Scenario 2</i>				
GraphSAGE-based	0.0580	0.0895	0.1330	0.2064
BERT-based	0.0523	0.0864	0.1347	0.2223
OneRec	<u>0.0835</u>	<u>0.1256</u>	<u>0.1806</u>	<u>0.2658</u>
BARGE	0.0976	0.1441	0.2045	0.2994

lated items. OneRec narrows the gap through semantic-ID-based generation, but its single-path token-level objective remains vulnerable to shallow-layer errors and off-path items. By combining item-level structure modeling, path-level reranking, and complementary decoding channels, BARGE delivers consistent improvements across both industrial scenarios. These results suggest that its structural adaptations generalize across different corpus scales and incumbent architectures.

4.2.4. Efficiency and Engineering Considerations

BARGE’s accuracy gains come at essentially zero marginal cost (Table 11). Using a 2-layer encoder instead of TIGER’s 4-layer one absorbs the parameters of ICA, the HPR scorers, and the DPD dual towers, so BARGE is actually *smaller* than TIGER (19.91 M vs. 22.71 M), while per-epoch train/infer cost grows by under 10% because the two DPD towers share the encoder and run in parallel.

Table 11 | Efficiency comparison between TIGER and BARGE on Amazon Beauty under identical configurations.

Method	#Params	Train (s/epoch)	Infer (s/epoch)
TIGER	22.71 M	22	17
BARGE	19.91 M	24	18

Beyond headline cost, four engineering invariants and recipes were key to shipping BARGE:

1. **Fixed evaluation budget.** Beam width stays at the TIGER-lineage default ($B = 20$) throughout offline evaluation, while the deployed retrieval channel runs 50 SID sequences per request (§4.2.5); HPR reranks within the beam and OR-fusion truncates at the same K , so no downstream capacity re-planning was needed.
2. **Robust hyperparameter plateaus.** The HPR fusion weight follows an inverted-U with a mechanistic explanation (rescue-vs-damage trade-off), optimal at $\lambda = 0.25$; the reranker pool saturates beyond Top- $N \approx 400$. Both settings transferred across datasets without re-tuning.
3. **Cheaper codebook by design.** The layer-wise decreasing (512, 256, 128, 64) codebook allocates capacity to the coarse layers where partitioning quality matters most, and uses *fewer* codeword embeddings (960 vs. 1024 for a uniform 4-layer configuration) while outperforming it.

4. **Orthogonality as a hard invariant.** The Householder parameterization enforces $R^\top R = I$ to $\sim 10^{-6}$ throughout training with no auxiliary loss, eliminating a class of regularization-tuning failure modes; the learned rotation reduces reconstruction loss by 0.04–0.05 over freezing $R = I$.

4.2.5. Deployment and Online A/B

Online serving. BARGE is integrated into Tencent’s commercial media platform as a generative retrieval channel. For each request, BARGE uses beam search to generate 50 SID sequences. Instead of mapping each SID sequence to a single item, we construct its query representation by summing the codebook embeddings corresponding to its constituent SID tokens, and use the resulting representation to retrieve relevant items from the item corpus through an embedding-based approximate nearest-neighbor retrieval service. Under the current per-request quota, each SID sequence retrieves 10 items, producing at most $50 \times 10 = 500$ candidates. These candidates are subsequently processed by the existing downstream cascade, including pre-ranking, fine-ranking, blending, filtering, and business-rule processing.

After cross-sequence deduplication, each request retains approximately 480 distinct candidates on average, corresponding to a duplication rate of only $\approx 4\%$. The limited overlap among the candidates retrieved from the 50 generated SID sequences indicates that the generation process does not collapse onto a small set of dominant items, but instead covers complementary regions of the item corpus. This candidate-level evidence suggests that BARGE maintains substantial retrieval diversity under a fixed beam budget.

Online A/B test. BARGE was trained on $2 \times$ NVIDIA H20 GPUs and evaluated in a two-week online A/B test, in which 6% of live traffic was allocated to the experimental group. All reported improvements are statistically significant. Compared with the deployed multi-stage incumbent, BARGE improved click-through rate by +0.60%, the number of unique clicking users by +1.34%, and total reading time by +1.70%.

Following the initial A/B test, we fully rolled out BARGE as a production retrieval channel and retained a long-term holdback group to continuously measure its incremental effect. Over the longer observation window, both click-through rate and total reading time exhibited larger and more stable improvements than those observed during the initial experimental period. One plausible explanation is the progressive reduction of the mismatch between the training and serving distributions. During the small-traffic experiment, many long-tail and heterogeneous items introduced by BARGE were under-represented in the historical exposure logs and could therefore be underestimated by downstream models. After the full rollout, the resulting exposure and user-feedback samples were gradually incorporated into the training pipeline, allowing the system to better model the item distribution introduced by BARGE and reducing the corresponding exposure bias.

The long-term holdback also captures cumulative system effects over a substantially longer period, while the larger observation window reduces the variance of the estimated treatment effect. Together, these factors help explain why the long-term gains are larger and more stable than the point estimates obtained from the short-term A/B test. The observation further indicates that the improvements introduced by BARGE are sustained after full production deployment.

Channel-level analysis. To further examine the contribution of BARGE at the retrieval stage, we deploy it as an individual retrieval channel and compare it with existing retrieval channels under the same traffic setting (Table 12). We report two metrics: the exposure adoption rate and CTR. The exposure adoption rate is defined as the ratio between the number of items from a retrieval channel that are eventually adopted and exposed by the downstream system and the total number of candidates submitted by that channel. It measures how effectively the

retrieved candidates pass through downstream pre-ranking, fine-ranking, blending, filtering, and business-rule processing.

As shown in Table 12, BARGE achieves the highest CTR of 9.93% among the four representative retrieval channels, compared with 8.70% for graph-based item-to-item retrieval (Graph-based I2I), 6.94% for DNN-based retrieval, and 5.61% for content-based retrieval (CB). Its exposure adoption rate reaches 1.02%, which is comparable to the 1.03% achieved by Graph-based I2I and places BARGE in the first tier among the evaluated retrieval channels. These results demonstrate BARGE’s effectiveness along two complementary dimensions: its retrieved items maintain a competitive pass-through rate through the downstream cascade, while achieving the highest click conversion rate once exposed. Therefore, BARGE provides measurable standalone value when deployed as an individual retrieval channel.

Table 12 | Online comparison of representative retrieval channels on Tencent’s commercial media platform.

Retrieval channel	Exposure adoption rate	CTR
Content-based (CB)	0.47%	5.61%
DNN-based	0.85%	6.94%
Graph-based I2I	1.03%	8.70%
BARGE	1.02%	9.93%

4.3. HiGR: Slate Generation for Industrial Recommendation

HiGR (**H**ierarchical **G**enerative slate **R**ecommendation) [24] is the slate-generation model in TGR-GenRec. Instead of generating one item at a time, HiGR treats the *entire ordered slate as a single generation unit*. This formulation matches how recommendations are actually presented on our surfaces: users consume an ordered list in one display, and its quality depends not only on the relevance of each item but also on list-level ordering, compatibility, and diversity. Existing SID-based generative recommenders in the OneRec lineage are designed for item-wise generation and therefore do not directly optimize these slate-level properties. HiGR closes this gap through a co-designed pipeline comprising a *prefix-contrastive tokenizer*, a *hierarchical slate decoder*, and *listwise preference alignment*, which jointly structure the SID space, generate slates efficiently, and align the resulting lists with holistic user preferences. Compared with OneRec under matched decoding settings (without KV caching), HiGR achieves more than a 5× inference speedup, and retains the best quality–efficiency trade-off once KV caching is enabled (§4.3.4). HiGR has been deployed across multiple Tencent commercial platforms, serving hundreds of millions of users and delivering consistent gains; the corresponding results are reported in §4.3.5.

4.3.1. Motivation: Three Disconnects Between Item Generation and Slate Optimization

Discriminative two-stage systems score candidates in isolation and then assemble a slate by greedy selection, heuristics, or reranking; optimizing items independently ignores inter-item dependencies and positional effects, yielding slates that are locally reasonable but globally sub-optimal. SID-based generative recommendation offers a more direct path: by generating an ordered slate autoregressively, it can jointly model item selection, ordering, and cross-item dependencies. However, extending item-wise SID generation to industrial-scale slate optimization exposes three disconnects:

- **(D1) Entangled SID space.** Generative slate quality hinges on item tokenization, but standard residual quantization produces entangled, sparse code spaces in which SID

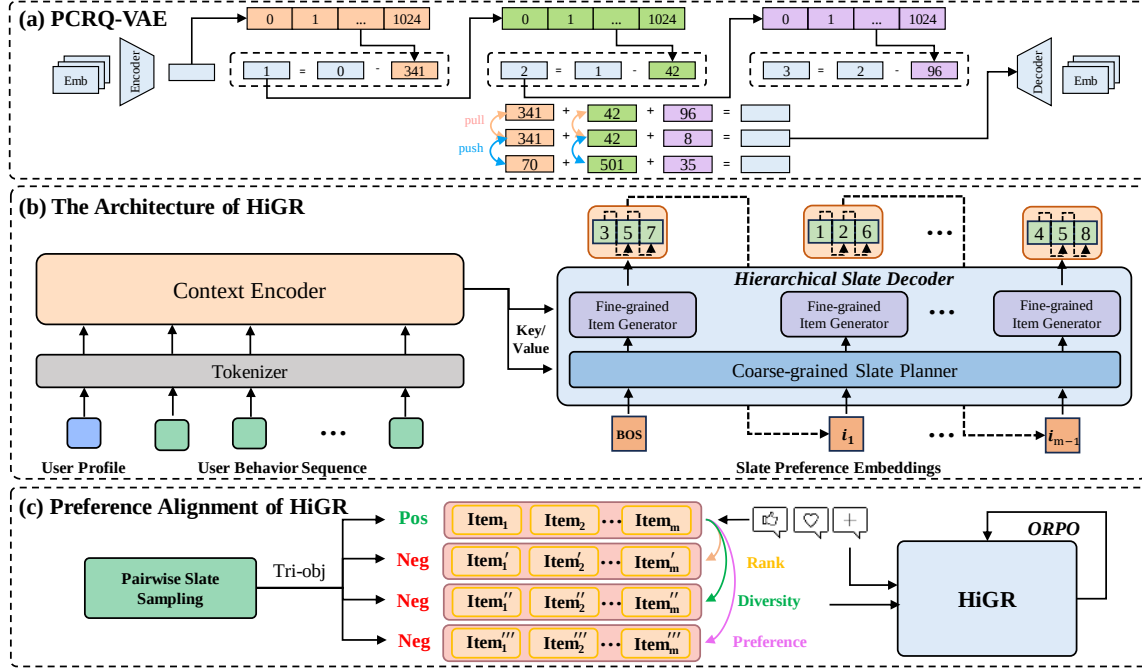


Figure 7 | Overall framework of HiGR: (a) PCRQ-VAE for semantic tokenization with prefix-level contrastive alignment over the first $D-1$ codebook layers; (b) Hierarchical Slate Decoder consisting of a coarse-grained slate planner (over preference embeddings) and a shared fine-grained item generator (over SIDs); (c) ORPO-based listwise preference alignment over triple-objective slate preference pairs (rank fidelity, genuine interest, diversity).

prefixes fail to reflect item semantics or collaborative relations. Inconsistent prefixes weaken controllability and make it hard to steer slate-level relevance and diversity through discrete codes.

- **(D2) Inefficient fine-grained decoding.** An M -item slate with D SIDs per item requires decoding $M \times D$ tokens. Full-sequence autoregressive beam search over this length is slow under real-time constraints and entangles intra-item semantic composition with inter-item transitions, obscuring the global slate structure.
- **(D3) Objective misalignment.** Token-level likelihood does not optimize user-perceived slate quality. Platforms care about multiple list-level objectives—ranking fidelity, genuine user interest, and intra-list diversity—none of which is captured by next-token prediction alone.

HiGR addresses these three disconnects through a coordinated design spanning the tokenizer, decoder, and training objective. PCRQ-VAE learns a structured hierarchical SID space for controllable slate generation (D1); HSD separates global slate planning from local item decoding and confines beam search to short item-level sequences (D2); and listwise ORPO aligns the generated slate with holistic user preferences (D3).

4.3.2. Method Overview

As shown in Figure 7, HiGR is a unified pipeline of three tightly coupled components—a prefix-contrastive tokenizer, a hierarchical slate decoder, and listwise alignment—each acting on one of the three disconnects. Given a user u with history S^u , the model generates an M -item slate $O^u = \mathcal{F}_\theta(u, S^u)$ optimized for holistic list quality.

PCRQ-VAE — structuring the discrete space (D1). The tokenizer keeps a standard RQ-VAE backbone but shapes the code space with two additions. A *global quantization* term aligns each item’s aggregated codeword with its latent, preventing residual vanishing in deep codebooks; and a *prefix-contrastive* objective pulls the first $D-1$ codebook layers of semantically or collaboratively related items together while excluding the leaf layer. High-level prefixes thus encode shared semantics and the final layer preserves item identity, so slate decoding can impose relevance and diversity constraints directly on discrete codes rather than on continuous embeddings.

HSD — coarse-to-fine generation (D2). Instead of autoregressing over the full $M \times D$ SID sequence, the hierarchical slate decoder factors generation into a coarse *slate planner* and a fine *item generator*. The planner autoregresses in a compact preference-embedding space—one embedding per slate position—to fix the global list intent, and a shared item generator then grounds each preference into a short local SID sequence; an intra-list diversity regularizer on the planned geometry keeps positions distinct. At inference the planner runs greedy while beam search is confined to the short per-item decoding, shrinking the effective search space and cutting decoding cost substantially versus full-sequence generation.

ORPO listwise alignment — optimizing slate quality (D3). To connect token-level generation with user-perceived quality, HiGR aligns whole slates via reference-free Odds Ratio Preference Optimization [38], which folds preference optimization into supervised learning and needs no reference model. Slate-level positive/negative pairs are mined from implicit feedback to cover three complementary objectives: *ranking fidelity* (against mis-ordered permutations), *genuine interest* (against negatively received items), and *diversity* (against over-similar lists). The supervised term preserves generative accuracy while the contrastive term rejects poorly ordered, irrelevant, or repetitive slates.

4.3.3. Offline Results

Public benchmark results. We construct chronological user sequences on KuaiRec and adopt a leave-five-out slate evaluation protocol. HiGR achieves the best result on every metric against traditional slate recommendation (ListCVAE [9]), discriminative sequential models (BERT4Rec and SASRec), and generative baselines (TIGER, HSTU, and OneRec), as shown in Table 13. Under the matched 25M-parameter setting, HiGR improves NDCG@5 from 0.1496 to 0.1574 (+5.2%) and Effective-View Recall@5 from 0.0422 to 0.0491 (+16.4%) over OneRec. Scaling HiGR to 100M parameters further raises these metrics to 0.1651 and 0.0546, corresponding to gains of 10.4% and 29.4% over OneRec-25M, respectively.

Industrial-scale offline evaluation. The industrial dataset is collected from Tencent’s commercial media platform and contains one billion pre-training samples, of which 3% are selected for ORPO post-training. We evaluate both exposure modeling and positive-feedback quality: Impression Hit@5 and Recall@5 measure whether the model recovers displayed items, Effective-View Hit@5 and Recall@5 focus on positively received items, and NDCG@5 measures their ordering quality. The compared systems span slate recommendation, discriminative sequential recommendation, and SID-based generative recommendation, providing a broad test across modeling paradigms.

At the same 25M-parameter scale, HiGR outperforms the strongest baseline, OneRec-25M, on all five industrial metrics. It improves Impression Hit@5 from 0.2438 to 0.2825 (+15.9%), Effective-View Hit@5 from 0.1603 to 0.1945 (+21.3%), and NDCG@5 from 0.0589 to 0.0714 (+21.2%). HiGR-25M without preference alignment already surpasses OneRec-25M on all ten metrics across the two datasets, isolating gains from the tokenizer and hierarchical decoder;

Table 13 | Offline performance on the industrial dataset and KuaiRec. Underlined and bold values denote the strongest baseline and the best overall result, respectively.

Method	Industrial dataset					KuaiRec				
	Impressions		Effective Views		NDCG@5	Impressions		Effective Views		NDCG@5
	Hit@5	Recall@5	Hit@5	Recall@5		Hit@5	Recall@5	Hit@5	Recall@5	
ListCVAE	0.0857	0.0178	0.0571	0.0117	0.0186	0.2304	0.0523	0.0757	0.0161	0.0630
BERT4Rec	0.0911	0.0187	0.0632	0.0129	0.0201	0.2612	0.0583	0.0829	0.0177	0.0713
SASRec	0.1057	0.0218	0.0700	0.0143	0.0243	0.3017	0.0756	0.1028	0.0231	0.0839
TIGER	0.1812	0.0383	0.1204	0.0249	0.0406	0.4455	0.1299	0.1566	0.0378	0.1363
HSTU	0.2281	0.0506	0.1487	0.0310	0.0492	0.5055	0.1334	0.1728	0.0397	0.1466
OneRec-25M	0.2438	0.0577	0.1603	0.0367	0.0589	0.5150	0.1395	0.1782	0.0422	0.1496
HiGR-25M w/o ORPO	0.2641	0.0612	0.1810	0.0395	0.0631	0.5218	0.1437	0.1805	0.0462	0.1536
HiGR-25M	0.2825	0.0692	0.1945	0.0433	0.0714	0.5290	0.1485	0.1846	0.0491	0.1574
HiGR-100M	0.3163	0.0760	0.2145	0.0495	0.0831	0.5360	0.1544	0.1905	0.0546	0.1651

ORPO then improves every metric further. The 100M model performs best overall, reaching 0.0831 NDCG@5 on the industrial dataset—a 41.1% gain over OneRec-25M—and demonstrating that the improvements continue with model capacity.

Component analysis and scaling. The ablations support the role of all three components. For PCRQ-VAE, applying contrastive alignment to the first $D-1$ codebook layers yields 2.37% collision, 93% concentration, and 66.47% consistency; extending the constraint to the leaf layer raises collision to 8.73%, confirming the need to preserve the last layer for item identity. For HSD, removing the ILD regularizer reduces intra-list diversity from 0.62 to 0.59 with nearly unchanged accuracy, whereas removing the context embedding drops NDCG@5 from 0.0753 to 0.0664 (both measured in the ablation configuration of [24], whose pre-alignment base differs from the 25M main setting); separate item generators provide no consistent gain over the shared design. For preference alignment, ORPO achieves ILD scores of 0.62 and 0.63 on the industrial dataset and KuaiRec, respectively, compared with 0.57 and 0.60 for DPO [39]. Ranking fidelity and genuine-interest objectives primarily improve NDCG@5, while the diversity objective primarily improves ILD; jointly optimizing all three produces the best NDCG@5 of 0.0831 while retaining an ILD of 0.62. Finally, experiments from 0.05B to 2B parameters follow a clear power-law trend in loss and NDCG@5 (Figure 8), indicating predictable gains from scaling model capacity—extending the LLM-style scaling behavior observed for generative ranking (§3.5) to whole-slate generation. The complete ablation tables and hyperparameter analyses are available in the original HiGR paper [24].

4.3.4. Efficiency and Engineering Considerations

Efficiency from hierarchical slate decoding. A flat slate generator represents an M -item slate as one sequence of MD SID tokens, where each item contains D codewords. This couples global slate modeling with fine-grained item construction and forces both training and inference to operate on a long target sequence. HSD factorizes the computation into an M -step slate planner over preference embeddings and a shared item generator over short D -token SID sequences. During training, this replaces target-side self-attention over one MD -token sequence with attention over the M planner positions plus a batch of short item-level sequences. The same item generator is shared across all slate positions, so its parameter count does not grow with slate length. HSD therefore improves the training-side computational structure and parameter efficiency, although the paper does not report a standalone training-speed measurement.

At inference, the efficiency gain is more direct. A OneRec-style decoder carries beam search through the entire MD -token slate, whereas HSD performs greedy autoregressive planning over only M preference embeddings and confines beam search to each short D -token item sequence.

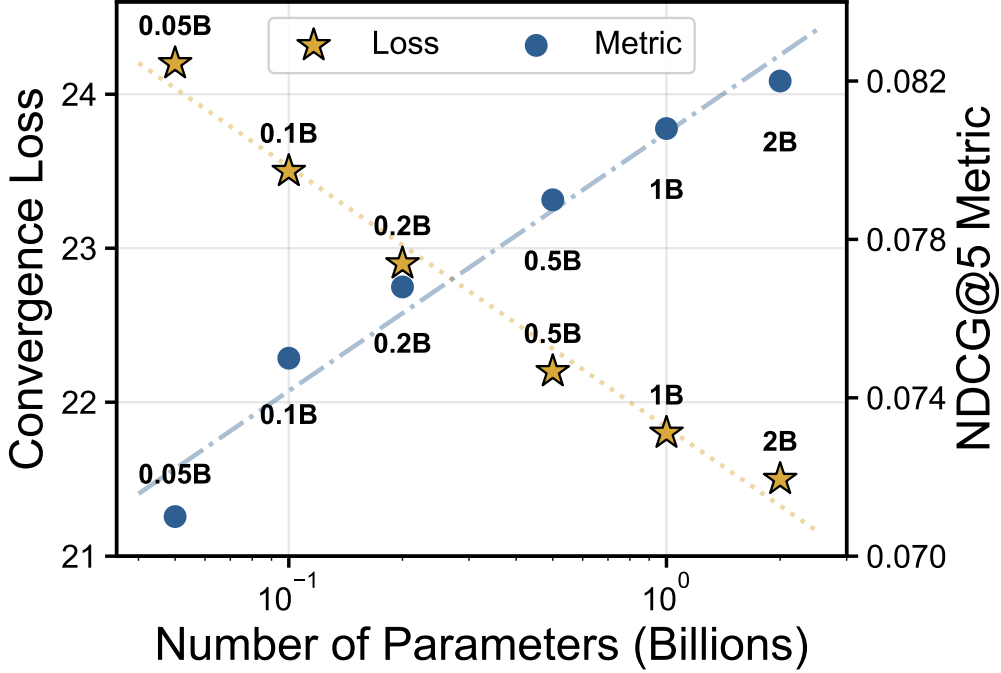


Figure 8 | Scaling behavior of HiGR from 0.05B to 2B parameters: training loss and NDCG@5 follow power-law trends, indicating predictable returns to model capacity for slate generation.

Its decoding complexity is reduced from $\mathcal{O}(BM^3D^3l_{\text{slate}}d)$ to $\mathcal{O}(M^3l_{\text{slate}}d + BMD^3l_{\text{item}}d)$, where B is the beam width, d the hidden dimension, and l_{slate} and l_{item} the depths of the planner and item generator (see Appendix D). Under a fixed total depth, allocating 14 layers to the planner and 2 to the item generator provides the best production trade-off: compared with a 12:4 split, it lowers latency by 26% with only a 0.6% relative decrease in NDCG@5. Without KV caching, HiGR achieves more than a $5\times$ inference speedup over OneRec-Beam (OneRec-style full-sequence decoding with beam search) while improving Recall@5 by over 5%; with KV caching, it retains the best quality–efficiency trade-off (Figure 9) [24]. In production serving, this decoding structure holds end-to-end P99 latency below 50 ms while reducing GPU demand by approximately 60% relative to a OneRec-style full-sequence autoregressive baseline at equivalent throughput—the serving envelope quoted in Table 7.

Beam-search optimization. We further optimize the local SID beam search through a Flash-Attention-aware tensor layout. At the first SID level, all beams share the same preference embedding, so the item generator runs only once and its top- B codewords initialize the beams, avoiding B identical decoder calls. At subsequent levels, all beam prefixes are decoded in one batched forward pass; cumulative scores over the $B \times V$ beam-codeword candidates are then flattened and pruned with a single top- B operation per request.

The main memory optimization comes from using two sequence layouts for the same flattened decoder states. For causal self-attention, they are interpreted as NB independent beam sequences, preserving isolation between beams, where N is the request batch size. For cross-attention, the B beams of each request are packed into one query sequence and attend to a single ragged user-history representation. This removes padding from variable-length histories and, more importantly, avoids replicating the encoder memory once per beam. The implementation therefore preserves exact beam-search scoring while reducing redundant first-step computation, Python-side beam iteration, and beam-dependent encoder-memory materialization.

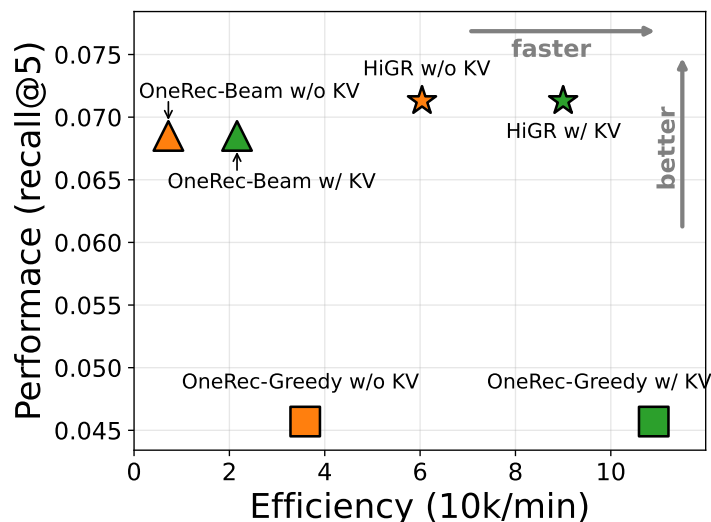


Figure 9 | Quality–efficiency trade-off of HiGR against OneRec-style decoding (beam search and greedy), with and without KV caching. HiGR attains the best Recall@5 at substantially higher decoding throughput.

Table 14 | Relative lift from online A/B experiments across three Tencent commercial recommendation scenarios.

Scenario 1				
Metric	Average	Average	Average	Average
Lift (%)	Stay Time	Watch Time	Video Views	Request Count
	1.03	1.22	1.73	1.57
Scenario 2				
Metric	CTR		Ad Revenue	
Lift (%)	0.68		0.56	
Scenario 3				
Metric	Average Watch Time		Average Video Views	
Lift (%)	1.14		0.88	

4.3.5. Deployment and Online A/B

HiGR has been deployed and evaluated through online A/B experiments in three Tencent commercial recommendation scenarios. Table 14 reports the relative lift over the corresponding incumbent production system in each scenario. Scenario 1 (5% live traffic) shows consistent engagement gains: Average Stay Time, Average Watch Time, Average Video Views, and Average Request Count increase by 1.03%, 1.22%, 1.73%, and 1.57%, respectively. Scenario 2 improves both user response and monetization, raising CTR by 0.68% and advertising revenue by 0.56%. Scenario 3 further improves Average Watch Time by 1.14% and Average Video Views by 0.88%. The gains across all three scenarios show that HiGR transfers beyond a single deployment setting and consistently improves engagement, traffic consumption, and commercial value.

5. TGR-Reason: Reasoning-Augmented Generative Recommendation

TGR-Reason is the reasoning-augmented direction of the TGR Stack. It turns LLM-derived intent inference into reasoning-enriched semantic IDs (*reason tokens*) that connect offline reasoning, memory retrieval, and online generative decision-making. A Think model trained with LatentRec [40] generates, outside the request path, a user’s top- K_r candidate items as complete SIDs. At serving time, these tokens follow two complementary paths. They directly provide semantic prompts to TGR-GenRec (§4) and act as queries for Reasoning-Guided Group Memory Retrieval (GMR), which retrieves relevant behavioral evidence from Group Memory. The retrieved group context is combined with the user’s Personal Memory—the observed behavior sequence—before being consumed by the generator. This design introduces no online reasoning rollout or additional autoregressive decision, and targets the requests for which co-occurrence provides the least evidence: cold-start, long-tail, and ambiguous-intent cases. In the deployed direct-injection configuration on one of Tencent’s commercial content platforms, reason tokens raise cold-start new-user Hit@1 from 0.0451 to 0.2606 (+477.8%), with consistent improvements across every user cohort (§5.5). All three modules operate in the shared SID space and reuse the tokenizer, encoder, and decoder interfaces of TGR-GenRec (principle P2, §2).

5.1. Motivation

The TGR-GenRec generator is trained to reproduce feedback-weighted co-occurrence. This works well when a user’s history densely constrains the next item. It falls short on the requests that matter most: new users with almost no history, long-tail items with sparse collaborative signal, and sessions of ambiguous intent. Such requests are underdetermined by co-occurrence alone. They call for external world knowledge and for the intent inference that a pattern-matching model never performs, and they are precisely where reasoning has the most to contribute.

This limitation also exposes a memory imbalance. A user’s observed sequence forms a precise but incomplete *Personal Memory*: it reflects individual interest, yet becomes uninformative when the history is short or ambiguous. Population-wide interaction sequences form a much richer *Group Memory*, but retrieving from them without an intent signal introduces popular yet irrelevant behavior. Reason tokens bridge the two. They infer plausible interests beyond the available Personal Memory and serve as semantic queries that select intent-consistent evidence from Group Memory, allowing population behavior to supplement rather than overwrite the individual’s own history.

Two lines of work make an LLM recommender reason, and both run it on the request path. Explicit reasoning writes the reasoning as language. OneRec-Think [10], for instance, produces a textual chain of thought before decoding each SID. This is interpretable, but it emits hundreds of extra tokens per request. Latent reasoning instead reasons in the hidden-state space. ReaRec [11] rolls the sequence representation forward for several steps before prediction, which removes the token cost but still runs the rollout at inference; STREAM-Rec [41] similarly moves sequential recommendation toward multi-step slow thinking at inference. Both regimes tie reasoning capacity to per-request computation, which industrial latency budgets cannot absorb. TGR-Reason moves reasoning off the request path entirely. Its Think model builds on the LatentRec framework [40], installs reasoning through training-time supervision, and periodically produces reason tokens in an offline pipeline. At serving time no LLM reasoning rollout is executed, so the reasoning cost is amortized rather than paid on every impression. To our knowledge, TGR-Reason is among the first to bring training-time-amortized reasoning to an industrial generative recommender [25].

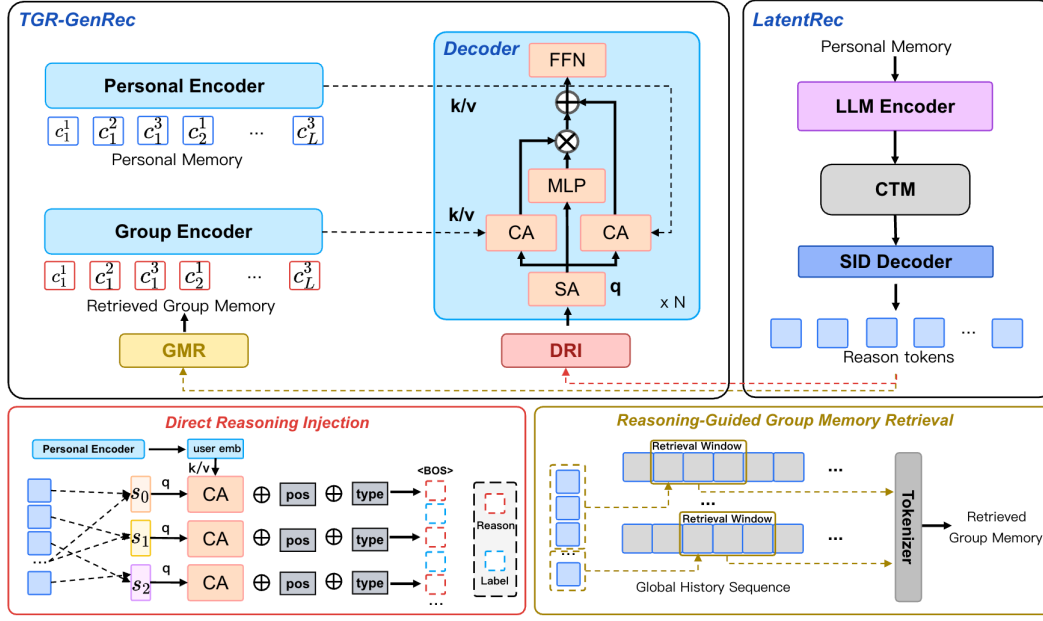


Figure 10 | TGR-Reason and memory augmentation in the middle layer of the TGR Stack. An offline reasoner produces reasoning-enriched SIDs that directly guide TGR-GenRec decoding through DRI and serve as token-level queries for GMR. TGR-Memory combines the GMR-retrieved collaborative evidence with the user’s Personal Memory and injects the resulting dual-memory context into TGR-GenRec.

5.2. Method Overview

Figure 10 organizes the framework into three cooperating modules. *TGR-Reason* generates reason tokens in an offline pipeline. Its Think model is built on LatentRec [40]: per-step supervision installs reasoning into the model parameters, allowing the deployed model to emit top- K_r complete SIDs through a standard decoding pass without an inference-time reasoning rollout. *TGR-Memory* maintains two complementary sources. Personal Memory is derived from the current user’s observed behavior and preserves individual interest; Group Memory indexes population-wide interaction sequences, from which GMR selects intent-consistent collaborative evidence using reason-token queries. *TGR-GenRec* consumes these signals through two interfaces: Direct Reasoning Injection (DRI) transforms reason tokens into level-aligned decoder representations, while GMR supplies retrieved Group Memory that is encoded and adaptively fused with Personal Memory. The two paths preserve the original number of generated SID tokens and introduce no additional autoregressive reasoning step. Group Memory retrieval and encoding remain bounded, non-autoregressive operations.

5.2.1. Reason Tokens as a Shared Reasoning–Memory Interface

A generative recommender can externalize reasoning in several forms, and that choice determines both how the signal enters the system and what must run on the request path. Table 15 contrasts four alternatives. Chain-of-thought text exposes the reasoning process but requires long autoregressive generation. Auxiliary generated features [42] introduce a separate representation and serving interface. Latent reasoning states avoid text generation, but remain internal to the model and cannot directly connect independently maintained reasoning, memory, and recommendation modules. Reason tokens instead express the Think model’s output as complete SIDs in the same vocabulary used throughout the TGR Stack.

This shared representation gives each reason token two complementary roles. As a *reasoning*

Table 15 | Alternative carriers for connecting reasoning with generative recommendation. Latent states are used internally during Think-model training, whereas offline-generated reason tokens form the shared interface to memory and online decoding.

Carrier	Produced	Request-path cost	Role in TGR-Reason
Chain-of-thought text	Inference time	High AR cost	Not adopted
Auxiliary generated features	Offline / online	Extra feature stack	Not used
Latent reasoning state	Training time	None at $K=0$	Internal supervision
Reason tokens (SIDs)	Offline	No reasoning rollout	Shared interface

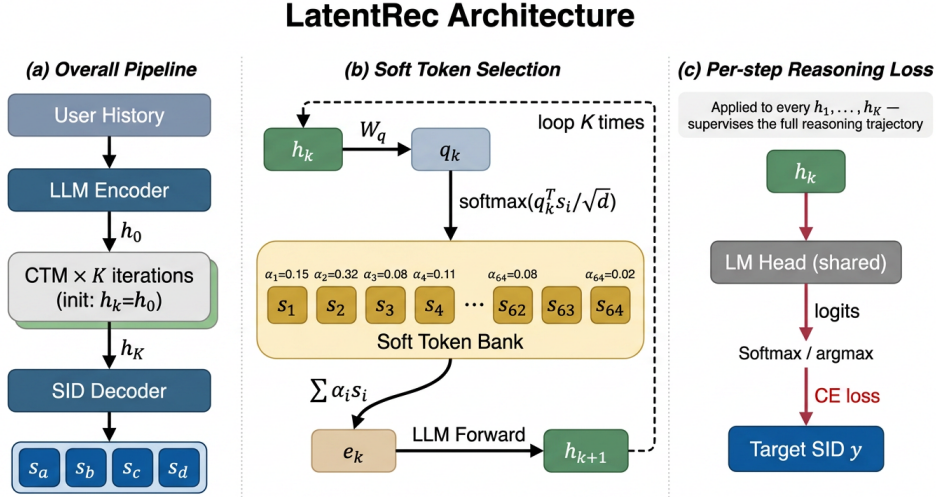


Figure 11 | LatentRec architecture inside the Think model. From the user-context hidden state h_0 , the Continuous Thought Module unrolls K latent steps. Soft Token Selection projects each hidden state back into the input-embedding distribution before it re-enters the backbone, and the Per-step Reasoning Loss (Eq. 3) supervises every intermediate state through the shared prediction head against the first level of the target semantic ID. The reasoning module runs only during training and is skipped at $K=0$ offline generation.

carrier, its layer-wise SID evidence directly prompts TGR-GenRec decoding. As a *memory query*, the complete SID drives GMR to retrieve intent-consistent behavior from Group Memory, which is subsequently combined with Personal Memory. No natural-language rationale crosses the module boundary, and no LLM reasoning rollout is added to online serving. The memory path incurs bounded retrieval and encoding work, but both paths preserve the existing SID tokenizer and autoregressive output space of §2. Latent reasoning remains an internal training mechanism, and the reason tokens are the primary interface shared across TGR-Reason, TGR-Memory, and TGR-GenRec.

5.2.2. Offline Reason Token Generation with LatentRec

The Think model is the offline reason-token generator of TGR-Reason, implemented as an SID-based generative recommender trained with our LatentRec framework [40]. LatentRec replaces inference-time reasoning, whether verbalized as chain-of-thought text or rolled out as latent states, with supervision applied only during training. The resulting model retains the reasoning gain while generating reason tokens through an ordinary SID-decoding pass in the periodic offline pipeline. Three components make this amortization possible: the Continuous Thought Module, Soft Token Selection (STS), and Per-step Reasoning Loss (PRL) (Figure 11).

Continuous Thought Module. The module produces the intermediate hidden states that per-step supervision later scores. Starting from the user-history hidden state $\mathbf{h}_0$, taken as the last-layer representation of the final history token, it unrolls K latent reasoning steps,

$$\mathbf{e}_k = \phi(\mathbf{h}_k), \quad \mathbf{h}_{k+1} = \text{LLM}(\mathbf{e}_k, \text{KV}^{(k)}), \quad (1)$$

where ϕ is the Soft Token Selection projection defined below and $\text{KV}^{(k)}$ is the key-value cache accumulated over the history tokens and the first k latent steps. Each latent step is a single-token forward pass that attends to this growing cache, so it reuses the history computation rather than recomputing it, and the rollout adds K such passes during training. The recurrence is deliberately minimal. It introduces no reasoning-specific Transformer weights, and its only trainable additions are the Soft Token Selection parameters, about 2.5M in total and under 0.2% of the backbone. Because the rollout feeds back through the same backbone under the same key-value cache, disabling it is exact rather than approximate. Setting $K=0$ during offline reason-token generation skips the module entirely and decodes directly from $\mathbf{h}_0$, which is the production configuration used here.

Soft Token Selection. A representation-space mismatch makes naive latent rollout unstable in the SID-decoding regime. In the Qwen3-1.7B backbone the last-layer hidden states carry roughly $14\times$ the L2 norm of the input embeddings. Feeding a raw hidden state back as the next input therefore drives the per-step predictions off the SID vocabulary as K grows, and the problem is sharper here than in math or logic latent reasoning because every intermediate state is also scored through the prediction head and must stay decodable. STS resolves this by never feeding a hidden state back directly. It maintains $N=64$ learnable soft tokens $\mathbf{S}$ initialized from the vocabulary embeddings and re-expresses each hidden state as a convex combination of them,

$$\alpha_i^{(k)} = \frac{\exp(\mathbf{q}_k^\top \mathbf{s}_i / \sqrt{d})}{\sum_j \exp(\mathbf{q}_k^\top \mathbf{s}_j / \sqrt{d})}, \quad \mathbf{e}_k = \sum_i \alpha_i^{(k)} \mathbf{s}_i, \quad (2)$$

with query $\mathbf{q}_k = W_q \mathbf{h}_k$. Since $\mathbf{e}_k$ is a convex combination of vocabulary-scale vectors, it stays within the input-embedding manifold in both norm and direction, whereas a scalar rescaling would fix the norm but leave the direction off-manifold. This keeps every projected input decodable by the shared prediction head, which is what lets the same head supervise all K steps. Empirically, the distinction is not cosmetic. Under matched settings, raw hidden-state feedback combined with per-step head supervision diverges within the first training epoch as the intermediate logits leave the SID vocabulary, so STS is necessary in this regime rather than a convenience.

Per-step Reasoning Loss. The third component supervises every intermediate state $\mathbf{h}_k$ through the shared prediction head, and it is the component that carries most of the gain. It adapts the progressive supervision of ReaRec [11], introduced for sequential-recommendation encoders, to the LLM SID-decoder regime by applying the shared prediction head at every latent step under a temperature that sharpens toward the last step,

$$\mathcal{L}_{\text{PRL}} = \frac{1}{K} \sum_{k=1}^K \text{CE}(\text{LM_head}(\mathbf{h}_k) / T_k, y), \quad T_k = T_b + T_s \left(1 - \frac{k}{K}\right), \quad (3)$$

where y is the first level of the target semantic ID and T_k decays linearly from a relaxed temperature at the first step to T_b at step K . The full objective is $\mathcal{L} = \mathcal{L}_{\text{CE}} + \lambda \mathcal{L}_{\text{PRL}}$ with $\lambda=0.1$. Supervising the first SID level gives the intermediate steps a stable target, since that level carries the coarsest routing decision in the RQ codebook. Supervising deeper levels would

instead require teacher-forcing the target SID prefix at each step, which couples the latent rollout to a fixed decoding path and conflicts with the autoregressive SID decoding used at inference, so those levels are left to $\mathcal{L}_{CE}$. Read this way, PRL is multi-step deep supervision through the prediction head, which is consistent with its gain surviving at $K=0$.

The property that makes reasoning amortizable is that this supervision improves accuracy even when generation uses $K=0$, where the module never runs in the forward pass. The gain lives in the LoRA adapters shaped by per-step supervision rather than in multi-step computation at generation time, so the reasoning module is discarded after training and the offline pipeline reduces to standard SID decoding. A chain-of-thought-distilled initialization is optional. It is not required at $K=0$, but it stabilizes deeper rollout, placing the value of multi-step latent computation at training time rather than on the production generation path. The production Think model therefore generates reason tokens with $K=0$.

Reason tokens. For each user refresh, the Think model outputs its top- K_r predicted items, where K_r denotes the number of retained reasoning candidates and each prediction is a complete L -level semantic ID. These *reason tokens* are the primary artifact that TGR-Reason exports: they directly guide TGR-GenRec decoding through DRI and serve as semantic queries for GMR. The Think model can additionally export a compact embedding that summarizes the reasoning behind each prediction, shown as the `cot_emb` feature in Figure 12, which the generator may consume as an auxiliary soft prompt alongside the SIDs. This embedding is precomputed during the offline refresh, so the request path still produces no natural-language reasoning, and the deployed configuration behind our results relies on the semantic-ID reason tokens alone. The notation K_r distinguishes the number of exported reason tokens from the K latent reasoning steps used only during LatentRec training.

5.2.3. Reasoning and Memory Injection into TGR-GenRec

The downstream generator reuses the shared SID tokenizer and encoder-decoder backbone of TGR-GenRec. For user u , let S^u denote the historical SID sequence that forms Personal Memory, and let the Think model provide K_r reason tokens, $\mathcal{R}_u = \{\mathbf{r}_{u,1}, \dots, \mathbf{r}_{u,K_r}\}$, where $\mathbf{r}_{u,k} = (r_{u,k}^1, \dots, r_{u,k}^L)$ is a complete L -level SID. As shown in Figure 10, these tokens enter TGR-GenRec through two complementary interfaces. DRI transforms their layer-wise semantic evidence into Personal-Memory-conditioned representations aligned with SID decoding. GMR instead uses the complete reason tokens to query Group Memory; the retrieved population evidence is then encoded and adaptively fused with Personal Memory.

Direct Reasoning Injection. DRI converts the K_r reason tokens into Personal-Memory-conditioned, layer-wise representations for hierarchical SID decoding. A single reason token describes a coherent candidate item, but its codewords play different roles along the coarse-to-fine SID hierarchy. DRI therefore reorganizes the reason-token set by codebook level while retaining all K_r alternatives. For level l , it looks up the shared SID embeddings

$$\mathbf{E}_u^l = [\mathbf{e}_l(r_{u,1}^l), \dots, \mathbf{e}_l(r_{u,K_r}^l)] \in \mathbb{R}^{K_r \times d}, \quad (4)$$

where $\mathbf{e}_l$ is the embedding table of the l -th codebook and d is the SID embedding dimension. No rank or candidate-position embedding is added: the reason tokens constitute an unordered evidence set rather than a reasoning sequence.

The relevance of this reasoning evidence depends on Personal Memory. Let $\mathbf{X}_u$ denote the N_u encoder-side context embeddings $[\mathbf{x}_{u,1}, \dots, \mathbf{x}_{u,N_u}]$ derived from the user’s profile and historical SID sequence, and let $m_{u,i}$ indicate a non-padding position. The block summarizes

Personal Memory with masked mean pooling,

$$\mathbf{q}_u = \frac{\sum_{i=1}^{N_u} m_{u,i} \mathbf{x}_{u,i}}{\max(1, \sum_{i=1}^{N_u} m_{u,i})}. \quad (5)$$

The same Personal-Memory query is then matched against the K_r reason-token codewords independently at each injected level. In particular, the l -th level has its own multi-head cross-attention parameters, with H heads of dimension d_h :

$$\mathbf{o}_{u,h}^l = \text{softmax} \left(\frac{(\mathbf{q}_u \mathbf{W}_{Q,h}^l)(\mathbf{E}_u^l \mathbf{W}_{K,h}^l)^\top}{\sqrt{d_h}} + \mathbf{A}_u \right) (\mathbf{E}_u^l \mathbf{W}_{V,h}^l), \quad (6)$$

$$\bar{\mathbf{z}}_u^l = \text{Concat}_{h=1}^H(\mathbf{o}_{u,h}^l) \mathbf{W}_O^l, \quad \mathbf{z}_u^l = \bar{\mathbf{z}}_u^l + \mathbf{p}_0 + \boldsymbol{\tau}_l. \quad (7)$$

Here, $\mathbf{A}_u$ is the additive attention mask over the K_r reason tokens (zero for available SIDs and $-\infty$ for missing ones), $\mathbf{p}_0$ is the item-position embedding, and $\boldsymbol{\tau}_l$ is the codebook-level type embedding. Because no candidate-order encoding is used, permuting the reason tokens leaves $\mathbf{z}_u^l$ unchanged. More importantly, the attention weights are conditioned on $\mathbf{q}_u$: rather than copying the top-ranked LLM prediction, the block learns which reasoning candidates are compatible with Personal Memory. Separate attention parameters across levels further allow coarse intent codes and fine-grained residual codes to select different evidence from the same reason-token set. If all reason tokens are missing, the resulting prompt positions are masked from the decoder.

The fused vectors $\mathbf{Z}_u = [\mathbf{z}_u^1, \dots, \mathbf{z}_u^{L_t}]$ act as continuous, hierarchy-aligned prompts, where $L_t \leq L$ is the number of injected levels. Let y^l denote the target item’s codeword at SID level l . The prompts are interleaved immediately before their corresponding target codewords:

$$[\text{BOS} \mid \mathbf{z}_u^1, y^1 \mid \dots \mid \mathbf{z}_u^{L_t}, y^{L_t} \mid y^{L_t+1}, \dots, y^L]. \quad (8)$$

A causal mask ensures that the decoder state at $\mathbf{z}_u^l$ predicts y^l without observing that target codeword; for levels beyond L_t , prediction follows standard autoregressive SID decoding. Meanwhile, every decoder state still cross-attends to Personal Memory, so the prompts augment rather than replace the user’s observed behavior. During beam search, $\mathbf{Z}_u$ is computed once before decoding and shared by all expanded beams. Consequently, direct reasoning injection exploits all K_r reason tokens without generating them online, changing neither the number of SID decisions nor the beam-search space.

Reasoning-Guided Group Memory Retrieval (GMR). Group Memory is an indexed corpus of population-wide behavior sequences, and each complete reason token acts as a semantic retrieval query over that memory. This complements DRI: DRI transfers the Think model’s inferred intent into decoding, while GMR grounds that intent in what users collectively consume afterward. Formally, let $\mathcal{G} = \{\mathbf{G}^{(v)}\}_v$ denote Group Memory, where $\mathbf{G}^{(v)} = [\mathbf{g}_1^{(v)}, \dots, \mathbf{g}_{T_v}^{(v)}]$ is user v ’s interaction sequence and every $\mathbf{g}_t^{(v)}$ is a complete L -level SID. For reason token $\mathbf{r}_{u,k}$, token-guided retrieval locates its occurrences across Group Memory and pools the W items immediately following each occurrence:

$$\mathcal{B}_{u,k} = \biguplus_{(v,t): \mathbf{g}_t^{(v)} = \mathbf{r}_{u,k}} \left\{ \mathbf{g}_{t+j}^{(v)} \mid 1 \leq j \leq W, t+j \leq T_v \right\}, \quad (9)$$

where $\biguplus$ denotes multiset union, retaining repeated successor occurrences across users and sequence positions for frequency counting. For each distinct successor SID, GMR counts its occurrences in the pooled multiset, ranks all successors by this frequency, and independently

retains the M most frequent ones for query $\mathbf{r}_{u,k}$. We denote the resulting ranked list by $\mathcal{C}_{u,k} = [\mathbf{c}_{k,1}, \dots, \mathbf{c}_{k,M}]$. Applying the same operation to all K_r reason tokens produces K_r query-specific successor lists, which are concatenated rather than globally reranked:

$$\mathcal{C}_u = \mathcal{C}_{u,1} \parallel \dots \parallel \mathcal{C}_{u,K_r}, \quad |\mathcal{C}_u| = K_r M. \quad (10)$$

Processing each reason token independently preserves the downstream behavior associated with every inferred intent: the forward window captures what users tend to consume after that SID, and the per-query top- M selection ensures that each reason token contributes M frequently observed subsequent items to the retrieved Group Memory context.

Dual-Memory Encoding and Adaptive Fusion. Each retrieved item $\mathbf{c}_{k,m}$ is represented by the same L -level SID tokenizer used for Personal Memory. Separate encoders then construct a Group Memory representation from the concatenated $K_r M$ retrieved items and a Personal Memory representation from the user’s observed history,

$$\mathbf{H}_u^g = \text{Enc}_{\text{group}}(\text{Tok}_{\text{SID}}(\mathcal{C}_u)), \quad \mathbf{H}_u^p = \text{Enc}_{\text{personal}}(\text{Tok}_{\text{SID}}(S^u)). \quad (11)$$

The shared SID space makes the two memories semantically compatible, while separate encoders preserve their different provenance: $\mathbf{H}_u^p$ records the current user’s individual interest, whereas $\mathbf{H}_u^g$ summarizes population-level successor behavior selected by the reason tokens.

At SID decoding level l , the self-attention state $\mathbf{d}_l$ queries Personal Memory and Group Memory through separate cross-attention operators,

$$\mathbf{a}_l^p = \text{CA}_p(\mathbf{d}_l, \mathbf{H}_u^p, \mathbf{H}_u^p), \quad \mathbf{a}_l^g = \text{CA}_g(\mathbf{d}_l, \mathbf{H}_u^g, \mathbf{H}_u^g). \quad (12)$$

Keeping the operators separate prevents population evidence from being mixed into Personal Memory before the decoder can assess its usefulness. A vector-valued gate adaptively activates Group Memory at each decoding level:

$$\mathbf{g}_l = \sigma(\mathbf{W}_g[\mathbf{d}_l; \mathbf{a}_l^p; \mathbf{a}_l^g] + \mathbf{b}_g), \quad \tilde{\mathbf{a}}_l = \mathbf{a}_l^p + \mathbf{g}_l \odot \mathbf{a}_l^g. \quad (13)$$

This asymmetric residual form keeps Personal Memory as the primary signal and activates Group Memory only where it is compatible with the current decoder state; noisy or weakly supported retrieval can therefore be suppressed without discarding individual context. Group Memory is encoded once per request and reused across decoding steps and expanded beams. Together, direct reasoning injection and dual-memory augmentation play complementary roles: the former supplies a Personal-Memory-conditioned semantic prior at the decoder input, while the latter grounds inferred intent in population behavior and exposes that evidence as auxiliary memory. Neither path increases the autoregressive SID decoding length or beam-search space.

5.3. Training Recipe

5.3.1. Think-Model Training

Offline Think model. Training the Think model proceeds in three stages that share one backbone. A vocabulary-expansion stage first adds the semantic-ID tokens to the backbone so that items become first-class symbols it can emit. A content-alignment stage then adapts the backbone to the target surface, tuning a small set of trainable tokens so that the model’s language is grounded in the platform’s items and their descriptions. A reasoning stage finally applies LatentRec: it trains only the LoRA adapters and the Continuous Thought Module under the per-step reasoning loss of §5.2.2, with $K_{\text{train}}=2$, loss weight $\lambda=0.1$, and temperatures $T_b=1$ and $T_s=5$, while the backbone stays frozen. Keeping the backbone frozen confines the reasoning signal to a light adapter, which is what lets the module be dropped at $K=0$ serving without disturbing the aligned backbone. The chain-of-thought-distilled initialization of §5.2.2 sits between alignment and reasoning and is used only when deeper rollout is required.

5.3.2. Gen-Model Training

The downstream TGR-GenRec model is trained with the same next-token-prediction objective used by the production SID generator. For each training request, the Think model’s top- K_r reason tokens are precomputed and treated as fixed conditioning features. DRI organizes their codewords by SID level, conditions them on Personal Memory, and inserts the resulting representations before the corresponding target codewords. Under teacher forcing, the generator predicts the target SID $\mathbf{y} = (y^1, \dots, y^L)$ one level at a time:

$$\mathcal{L}_{\text{gen}} = - \sum_{l=1}^L w_l \log p_{\theta} \left(y^l \mid y^{<l}, \mathbf{H}_u^p, \mathcal{R}_u \right), \quad (14)$$

where w_l assigns larger weights to coarser SID levels. Gradients update the generator and DRI parameters, but do not propagate into the nearline Think model.

When GMR is enabled, its retrieved context $\mathcal{C}_u$ is added to the conditioning variables, and the Group Encoder and adaptive fusion gate are optimized jointly through the same NTP loss. GMR itself is non-parametric and receives no gradient. No additional reasoning-specific objective is introduced on the Gen-Model side; reason tokens and dual-memory context improve generation by conditioning the standard SID prediction task.

5.3.3. Offline Generation at Scale

Treating reasoning as an offline feature is practical only if reason tokens can be regenerated for the whole user base on a routine schedule and within a fixed compute budget. Two design choices make this possible. A two-pipeline refresh decouples how often the Think model is retrained from how often its predictions are produced, and a batched decoding scheme runs the per-user top- K_r generation cheaply on an unmodified inference engine.

Routine production. The refresh runs on two cadences that serve different purposes. A weekly pipeline retrains the Think model end to end so that vocabulary expansion and content alignment keep tracking new items and shifting user interests, and it produces one versioned reasoning model each week. A daily pipeline then applies the current model to the users a staleness detector marks for refresh and writes their top- K_r reason tokens into the Reason Store that the generator reads. Separating the two cadences keeps the expensive retraining weekly while the lighter prediction pass stays daily, so freshness and cost can be tuned independently. Retraining on a fixed-size weekly sample rather than the full log holds each cycle within budget without sacrificing the downstream gain, and every weekly model is versioned so that the generator always consumes a complete and self-consistent release rather than a partially updated one.

Offline decoding at scale. The daily pass is the cost bottleneck, since it must emit per-user top- K_r semantic IDs for tens of millions of users, which amounts to a constrained top- K_r beam search over a short fixed-length code. Neither native path of a high-throughput inference engine (vLLM) fits this shape. The sampling path, even behind a trie logits filter, replaces beam enumeration with stochastic draws and loses top- K_r recall, dropping Hit@5 well below the reference decoder. The native beam-search path preserves validity but runs as an unoptimized loop that decodes slower than the eager baseline, which removes the very speedup that motivated the migration. The mismatch is structural, because a semantic ID is a short code drawn from a constrained vocabulary, exactly the regime that the engine’s throughput machinery is not built to enumerate.

Rather than modify the engine, we recast beam search as a fixed number of batched single-token rounds that its scheduler already runs well. A semantic ID spans a fixed number of codebook levels, so decoding completes in that many rounds. Each round expands every live beam by its top- B next tokens in one batched forward pass and prunes back to the B best partial SIDs by cumulative log-probability, which is the same deterministic enumeration as standard beam search rather than an approximation. Continuous batching packs the beams of all users in a cycle into each pass, and prefix caching stores the long shared user-context prefix once, so the marginal cost of a round is a single incremental decoding step. The SID trie serves as a final validity check instead of a per-step callback, since the enumerated beams are already valid in practice, which keeps the fast path free of per-token masking. The speedup therefore comes entirely from the engine’s batching and cache reuse under this decoding schedule, not from any change to its kernels or scheduler.

Because the rounds reproduce the reference enumeration exactly, the schedule matches the reference decoder’s Hit@5 bucket by bucket at the same beam width, with the cold, warm, and hot buckets all matching or slightly exceeding the baseline, and it emits only valid semantic IDs. The speedup thus carries no accuracy cost. On a production cluster of eight A100 GPUs it shortens the end-to-end generation cycle by $2.2\times$, from 174 to 80 minutes, which brings a ten-million-user refresh from about 2.9 days down to 1.3. That headroom is what lets the daily pipeline run within budget and keeps reasoning an offline-amortized feature rather than a per-request cost.

5.4. Deployment

Asynchronous Reason Materialization. To avoid placing expensive Think-model inference on the critical serving path, TGR-Reason materializes user-level reasoning outputs as reusable *Reason records* in the Reason Store. Request-triggered nearline jobs refresh selected users under gain, freshness, and budget constraints, while scheduled offline jobs backfill missing, stale, failed, or version-incompatible records. Both paths update the store conditionally, ensuring that newly materialized records are consumed only by subsequent requests. Throughout this section the architecture is described in its full design, including the GMR and `cot_emb` paths; the deployed default configuration behind the results of §5.5 injects the semantic-ID reason tokens through DRI alone (§5.2.3).

5.4.1. Serving Efficiency

Moving reasoning off the critical path. TGR-Reason decouples expensive user-level reasoning from latency-critical candidate generation. The Think model may contain billions of parameters and require substantially more computation than the online generator. Running it synchronously for every request would incur prohibitive latency and GPU cost. TGR-Reason instead materializes its outputs as reusable user-level Reason records, keeping the Think model outside the synchronous request path and avoiding new reasoning-trace generation for the current response.

At serving time, the online generator reads the most recent valid Reason record through a key-value point lookup. The pre-materialized DRI representation is injected into the decoder side of TGR-GenRec as a fixed-size soft prompt. In parallel, the stored complete reason SID tokens are used by GMR to retrieve a bounded set of group-level behavioral memories. The retrieved memories are pooled and incorporated into the context-encoding stage before SID decoding. DRI and GMR thus provide complementary conditioning signals: DRI transfers user-specific reasoning intent directly to generation, whereas GMR grounds generation in population-level behavioral evidence.

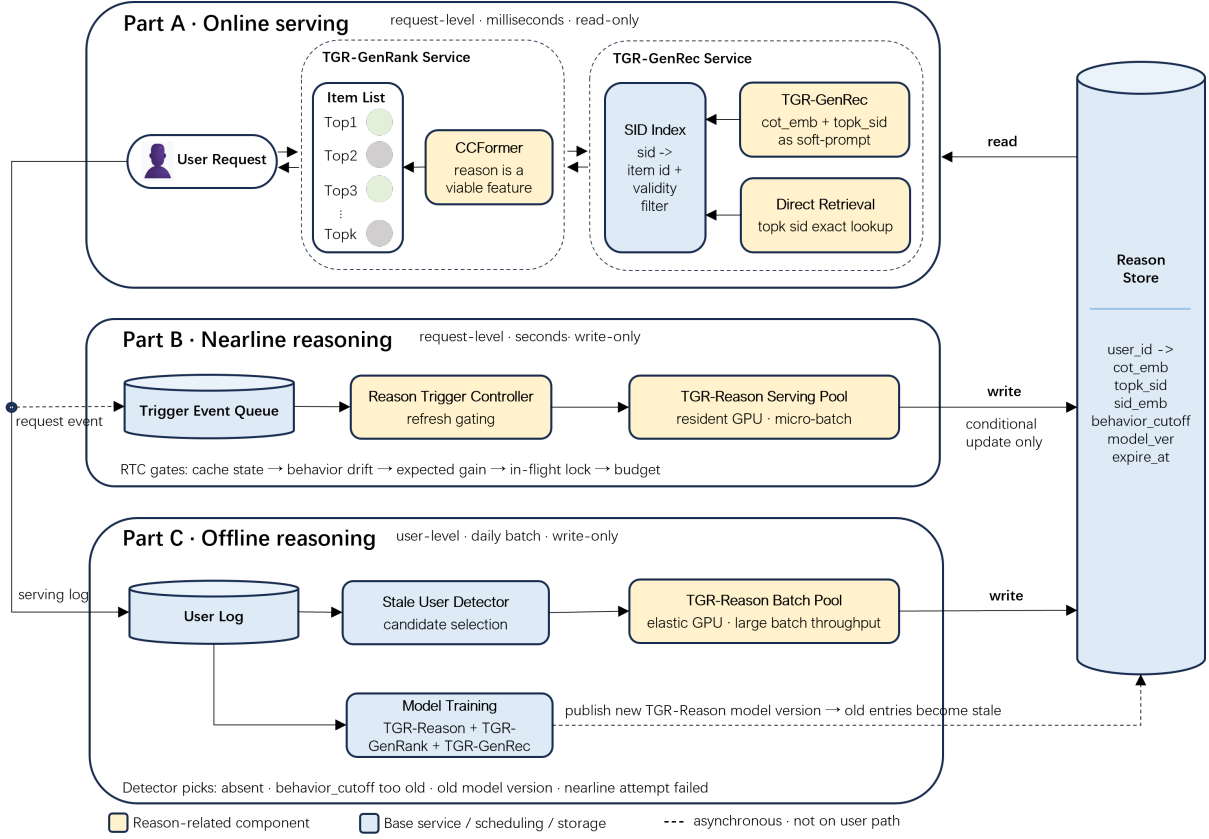


Figure 12 | The three-tier deployment architecture of TGR-Reason. **Part A: Online serving** operates at request level under a millisecond latency budget. It reads a pre-materialized Reason record from the Reason Store and uses the record for both reason-enhanced generation and direct SID retrieval. **Part B: Nearline reasoning** is asynchronously triggered by online requests. The Reason Trigger Controller (RTC) admits only refreshes with sufficient expected utility, which are executed by a resident-GPU serving pool and conditionally written back to the store. **Part C: Offline reasoning** periodically detects missing, stale, failed, or version-incompatible users and regenerates their Reason records using a throughput-oriented batch pool; a separate scheduled job retrains and publishes the associated models. The online tier is read-only, whereas the nearline and offline tiers update the shared Reason Store through version-aware conditional writes.

The stored SID predictions additionally support a decoder-free direct retrieval path. Specifically, `topk_sid` is resolved through the SID index to obtain exact item candidates, after which invalid, unavailable, or out-of-domain items are filtered. Unlike DRI and GMR, which condition TGR-GenRec, direct retrieval directly produces candidate items and does not participate in autoregressive decoding. The generated and directly retrieved candidates are assigned independent quotas before being returned to the downstream ranking stage.

DRI injection, GMR retrieval and pooling, and SID-index lookup are all bounded non-autoregressive operations. They add neither new reasoning-token generation nor additional SID decoding steps, and therefore leave the SID decoding horizon, beam width, and beam-search space unchanged relative to the original TGR-GenRec serving path. The additional request-time cost is limited to Reason-Store access, metadata validation, fixed-size feature projection, bounded Group-Memory retrieval and aggregation, and SID resolution, rather than another round of Think-model reasoning. The expensive reasoning cost is instead amortized over subsequent requests through budget-controlled nearline refreshes and offline backfills.

5.4.2. Online Serving

Part A: online candidate generation and ranking. Upon receiving a user request, the online service retrieves the user’s conventional sequence, profile, and contextual features. In parallel, it performs a point lookup in the Reason Store using the user identifier. A Reason record is considered viable only if it is present, unexpired, supported by the deployed model and feature schema, and associated with the active SID-codebook version.

When the record is viable, its DRI representation and reason SID tokens are consumed by TGR-GenRec through complementary conditioning paths. The DRI representation is injected into the decoder as a fixed-size soft prompt, while GMR uses the complete reason SID tokens to retrieve group-level memory and incorporates the resulting memory representation into the context encoder before SID decoding. TGR-GenRec then autoregressively generates a SID list. In parallel, the direct retrieval path performs an exact lookup of the stored `topk_sid`. The SID index maps the predicted SIDs to item identifiers and applies validity and availability filters. Candidate items from the generative and direct retrieval paths are assigned separate source tags and quotas before being passed to the downstream ranking stage.

Reason features are also reused by the ranking model. In particular, the valid Reason record is projected into rank-side features and provided to CCFormer in TGR-GenRank, complementing the conventional user, item, and context features. Thus, the same materialized Reason record improves both candidate generation and downstream ranking without requiring synchronous Think-model inference.

A Reason-store miss or an invalid record never blocks the request. TGR-GenRec falls back to its original no-Reason generation path, the direct retrieval path is skipped and its quota is filled by baseline candidate sources, and TGR-GenRank uses its base CCFormer feature set without Reason-derived features. Thus, the availability of the online recommendation service does not depend on successful execution of the Think model.

Part B: request-triggered nearline refresh. Every incoming request may asynchronously emit a lightweight trigger event, such as `user_id`, request time, and scene identifier, to the Trigger Event Queue. This event emission follows a fire-and-forget pattern and proceeds in parallel with the online candidate-generation path; the current request neither waits for nor consumes the newly generated Reason record.

The RTC consumes these events and determines whether recomputing the user’s Reason record is worth the associated inference cost. Its admission policy jointly considers:

1. whether a compatible Reason record is missing or approaching expiration;
2. the degree of user-behavior change since the previous `behavior_cutoff`;
3. the expected marginal recommendation gain for the current user or user segment;
4. user-level cooldown and single-flight constraints;
5. the current inference budget, queue backlog, and GPU capacity.

Only admitted jobs are submitted to the resident-GPU TGR-Reason serving pool. The pool uses dynamic micro-batching to improve utilization while maintaining second-level freshness. Its output is validated and conditionally written to the Reason Store. The refreshed record becomes visible to subsequent requests rather than retroactively affecting the request that triggered the update.

Part C: offline backfill and model update. The offline tier operates at user granularity and prioritizes throughput rather than individual-job latency. A scheduled stale-user detector scans the user log and Reason-store metadata to identify:

- users with no materialized Reason record;
- users whose behavior has advanced beyond the stored `behavior_cutoff`;
- expired or low-quality records;
- failed or skipped nearline refreshes;
- records produced by an obsolete Think-model, prompt, feature, or SID-codebook version.

The selected users are processed by an elastic large-batch reasoning pool, which maximizes GPU throughput and writes the regenerated records through the same conditional-update interface used by the nearline tier. In our deployment, this backfill is executed daily, whereas model retraining and checkpoint publication follow a lower-frequency schedule, e.g., weekly. Publishing a new model version marks incompatible Reason records as stale and triggers their progressive reconstruction through the nearline and offline refresh mechanisms.

5.4.3. Store Consistency and Failure Isolation

Versioned Reason records. The Reason Store is a distributed key-value abstraction backed by a managed key-value service on Tencent Cloud, rather than being tied to a particular storage engine. Each record is keyed by the user identifier, optionally augmented with the recommendation scene and experiment variant, and contains

$$\{\text{cot_emb}, \text{topk_sid}, \text{sid_emb}, \text{behavior_cutoff}, \text{generated_at}, \text{model_version}, \text{sid_codebook_version}, \text{expire_at}\}. \quad (15)$$

The underlying managed service provides low-latency point lookup, TTL-based expiration, and atomic conditional-update semantics. This design keeps the serving interface independent of a specific storage product while allowing the storage tier to be selected according to access density, capacity, latency, and cost requirements.

Conditional updates. Nearline and offline jobs may finish out of order. For example, an offline job based on user behavior observed up to 18:00 may complete after a nearline job based on behavior observed up to 20:00. Comparing only `generated_at` would incorrectly allow the later-finishing but semantically older record to overwrite the fresher one.

TGR-Reason therefore accepts only records compatible with the active model and feature versions. Within the same version, an incoming record may replace the stored record only when its `behavior_cutoff` is newer. The `generated_at` field is retained for monitoring and auditing, but is not used as the primary freshness order. This version-aware conditional-write policy prevents stale offline jobs from overwriting newer nearline results.

Failure isolation. The three tiers are deliberately failure-isolated. A failure in the trigger queue, RTC, or nearline serving pool only reduces Reason freshness. A failure in the offline batch pool only delays coverage recovery. Neither failure mode interrupts the online request path, which can continue with the most recent compatible Reason record or fall back to no-Reason TGR-GenRec. This separation allows the system to trade reasoning freshness against inference budget without coupling large-model availability to the availability or tail latency of online recommendation.

Table 16 | Offline comparison between reason-token injection and the online TGR-GenRec baseline on one of Tencent’s commercial content platforms. Results are averaged with equal weight over three test dates and use the deployed default configuration (Personal Encoder + Direct Reasoning Injection). The GMR branch is excluded. Relative gains are shown in italics.

Cohort	Model	Hit@1	Hit@5	Hit@10	Hit@20	Hit@50
All rec. traffic	TGR-GenRec (online)	0.3385	0.5152	0.5792	0.6377	0.7063
	TGR-Reason	0.4339	0.5686	0.6155	0.6635	0.7233
	<i>Rel. gain</i>	<i>+28.2%</i>	<i>+10.4%</i>	<i>+6.3%</i>	<i>+4.1%</i>	<i>+2.4%</i>
Primary rec. traffic	TGR-GenRec (online)	0.2247	0.4306	0.5116	0.5859	0.6671
	TGR-Reason	0.3563	0.4870	0.5461	0.6085	0.6847
	<i>Rel. gain</i>	<i>+58.6%</i>	<i>+13.1%</i>	<i>+6.7%</i>	<i>+3.9%</i>	<i>+2.6%</i>
Primary rec., cold-start	TGR-GenRec (online)	0.0451	0.3379	0.4551	0.5570	0.6949
	TGR-Reason	0.2606	0.4702	0.5684	0.6687	0.7753
	<i>Rel. gain</i>	<i>+477.8%</i>	<i>+39.2%</i>	<i>+24.9%</i>	<i>+20.1%</i>	<i>+11.6%</i>

5.5. Evaluation Results

Offline results. Table 16 compares TGR-Reason with the deployed production generator on one of Tencent’s commercial content platforms. We report Hit@ K at five cutoffs over three progressively more focused cohorts: all recommendation traffic, primary recommendation traffic, and cold-start users within the primary recommendation traffic. TGR-Reason improves every cutoff in every cohort, showing that the reasoning signal is useful not only for the top-ranked prediction but throughout the retrieved candidate set.

The improvement is strongest near the head of the list. Across all recommendation traffic, Hit@1 rises from 0.3385 to 0.4339 (+28.2%), while Hit@5 and Hit@50 improve by 10.4% and 2.4%, respectively. Restricting evaluation to the primary recommendation traffic increases the Hit@1 gain to 58.6% (0.2247 $\rightarrow$ 0.3563), showing that the benefit of reason-token injection is especially pronounced in the platform’s primary recommendation traffic. The gain narrows as the cutoff K grows because the production model already recovers many relevant items in a wider candidate set; reasoning contributes most by moving a plausible target toward the first few positions.

Cold-start users within the primary recommendation traffic exhibit the largest improvement. Their Hit@1 increases from 0.0451 to 0.2606, an absolute gain of 0.2155 and a relative gain of 477.8%. The effect remains substantial at larger cutoffs: Hit@5 improves by 39.2%, Hit@10 by 24.9%, Hit@20 by 20.1%, and Hit@50 by 11.6%. This pattern supports the central motivation of TGR-Reason: when behavioral history is sparse, a co-occurrence-driven generator has little evidence with which to resolve intent, whereas reasoning-enriched tokens provide a semantic prior that narrows the target space. As user history and candidate-set width increase, collaborative evidence recovers and the marginal contribution of that prior naturally becomes smaller.

Online A/B. We further evaluate TGR-Reason in a live online A/B experiment on the same Tencent commercial content platform, using the incumbent production generator as the control. As shown in Table 17, TGR-Reason improves the Effective Consumption Rate by 1.75% and the Exposure-to-Conversion Rate for new users by 13.09%; both lifts are statistically significant. The first result shows that the offline gain translates into more exposures producing meaningful content consumption rather than merely increasing candidate recall. The substantially larger new-user conversion lift is also consistent with the offline cohort analysis: reasoning-enriched SIDs supply useful semantic evidence precisely where personal interaction history is least informative. Together, the online results confirm that the cold-start improvements in Table 16

Table 17 | Online A/B results of TGR-Reason on one of Tencent’s commercial content platforms. Relative lifts are measured against the incumbent production generator; both improvements are statistically significant.

Metric	Relative lift	Significance
Effective Consumption Rate	+1.75%	Significant
Exposure-to-Conversion Rate (New Users)	+13.09%	Significant

survive the complete production cascade and produce measurable user-value gains.

6. Conclusion, Limitations, and Future Directions

6.1. Conclusion

In this report we introduced **TGR**, an industrial-scale generative recommendation stack consisting of three coupled directions: TGR-GenRank (generative-paradigm ranking), TGR-GenRec (unified end-to-end generative recommendation), and TGR-Reason (reasoning-augmented generative recommendation). The three directions rest on shared foundations: the generative-recommendation models share a semantic-ID tokenizer family (PCRQ-VAE for slate generation, OSQ-VAE for NTP generation) and a common user/context encoding interface; the whole stack is trained and served on the Numerous-Torch infrastructure. TGR-GenRank is realized by CCFormer, which unifies feature-field separated cross attention with subspace token mixing and hierarchical sequence compression: it consistently outperforms strong industrial baselines offline, follows predictable scaling laws in sequence length and model size at roughly half the GFLOPs of HSTU, trains $2.21\times$ faster than HSTU, and is fully deployed in a video-recommendation scenario (+3.57% CTR) and an advertising-ranking scenario (+1.71% advertising revenue). The most fully developed component, TGR-GenRec, spans one production model per generation paradigm: BARGE, which structurally adapts next-token-prediction SID generation through item context-aware attention, hierarchical path reranking, and dual-path decoding at zero additional serving cost; and HiGR, which realizes whole-slate generation by combining Prefix-Contrastive RQ-VAE, a Hierarchical Slate Decoder, and ORPO-based listwise multi-objective preference alignment. TGR-Reason is realized by a LatentRec-trained Think model whose reason tokens, exported offline, are injected into the production generator through Direct Reasoning Injection at zero request-path reasoning cost, raising cold-start new-user Hit@1 by +477.8% and delivering statistically significant online gains (+1.75% Effective Consumption Rate, +13.09% new-user Exposure-to-Conversion Rate). TGR has been deployed on multiple Tencent surfaces, serving hundreds of millions of users.

6.2. Limitations

The limitations below reflect where the stack stands today.

- **A stack of coupled models, not yet one model.** TGR upgrades or replaces the cascade piece by piece: TGR-GenRank still emits per-item multi-task scores that the downstream cascade consumes, and TGR-GenRec currently ships two production models with two tokenizers (OSQ-VAE for BARGE, PCRQ-VAE for HiGR).
- **Cold-start items.** The hierarchical SID space inherits the cold-start limitations of vector-quantized tokenizers: a brand-new item enters the code space on semantic evidence alone, before the collaborative relations that shape SID prefixes (§4.3.2) can be estimated for it. TGR-Reason’s demonstrated cold-start gains concern new *users* (§5); cold-start *items* remain governed by tokenizer quality.

- **Global alignment trade-offs.** ORPO-based alignment integrates ranking fidelity, genuine user interest, and diversity under a single global weighting (§4.3.2), uniform across users and contexts; a personalized or context-conditioned weighting of the three listwise objectives remains open.
- **Reasoning is amortized, not deliberative.** TGR-Reason spends all reasoning off the request path: the Think model is retrained weekly, reason tokens are regenerated daily (and, for trigger-gated users, within seconds in the nearline tier), and serving reads tokens precomputed at $K=0$. Intra-session intent shifts therefore reach the generator only through its own input context, and explicit in-text reasoning remains too expensive for the request path.

6.3. Future Directions

1. **Bending the scaling curve, not just riding it.** The scaling frontier of generative ranking has moved from demonstrating scaling laws [6] to improving their slope: ULTRA-HSTU reports over $5\times$ faster training scaling and $21\times$ faster inference scaling through end-to-end model-system co-design [43]. TGR will push HiGR beyond the 0.05B–2B range in which its scaling law has been observed (§4.3.3)—including a unified lazy-decoder realization of planner and generator—and pursue the same co-design discipline (sparse attention, input-sequence and topology design) on Numerous-Torch, always inside the < 50 ms P99, fixed-GPU envelope that keeps TGR-GenRec deployable (§4.3.4).
2. **Reasoning-native generation and test-time compute.** Reasoning is becoming a first-class citizen of industrial recommendation rather than an offline add-on: OneRec-Think brings in-text chain-of-thought into the generator [10], ReaRec explores latent multi-step reasoning at inference [11], and RecGPT-V3 internalizes verbose rationales into compact latent tokens backed by a stateful memory hub [44]—independent convergence on the latent, amortized route that TGR-Reason takes (§5). TGR will extend this route with selective request-time deliberation ($K>0$ decoding for high-value or cold-start requests), distillation into latency-tolerant variants in the spirit of reasoning-augmented LLMs [14], and richer memory beyond the current Personal/Group split.
3. **Agentic self-evolution of the recommender.** Self-evolution has moved from vision to production practice: LLM agents that autonomously propose, screen, and A/B-validate improvements to a large-scale industrial recommender—spanning optimizers, architectures, and reward functions—have been shown to surpass conventional engineering workflows in development velocity and model performance [45], and surveys chart the same trajectory for agents at large [46]. TGR will likewise advance along this self-evolution direction in its future iterations.

Appendix

A. Contributions

The listing of contributors is in alphabetical order based on their last names.

Lei Cheng
Haonan Hu
Beibei Kong
Yudong Li
Zang Li

Yunsheng Pang
Hongyang Su
Jianchao Tu
Yunlong Wang
Bing Wen

Junzhang Zhu
Shaojie Zhu
Chengxiang Zhuo

B. Notation

Table 18 summarizes the stack-level notation used throughout the report, followed by the principal model-specific symbols of each technical chapter. Because the three production models are presented as independent, parallel works (§4) and follow the notation of their source papers, symbols below are scoped to the sections listed, and the same letter may denote different quantities in different chapters—e.g., L is the per-item SID depth in §4.2 and §5 (denoted D elsewhere in the report) but the number of interaction blocks in §3. Symbols not listed here are local to the subsection in which they appear.

C. Default Configurations and Hyperparameters

Table 19 consolidates the default configurations of the three production models and TGR-Reason as a single reference, drawn from the corresponding source papers [22–24] and §5; the body sections describe the context in which each setting is used. Values not published by the source papers (e.g., the ILD regularization weight β of HiGR) are omitted.

D. Complexity Analysis

CCFormer (§3). CCFormer avoids the $\mathcal{O}(L_s^2)$ cost of global self-attention over the behavior sequence by construction: relative temporal-positional encoding is applied within local groups of m behaviors at $\mathcal{O}(L_s m)$ cost, subspace token mixing processes the sequence at cost linear in L_s , and the strided convolution after each block hands a geometrically shorter sequence to the next block, so deeper blocks retain access to the full history at a fraction of its length. At matched capacity this yields roughly half the per-sample GFLOPs of HSTU (Table 4).

BARGE (§4.2). In hierarchical SID beam search, inference latency scales approximately linearly with the beam width B and decoding memory grows with $B \times L$, which is why BARGE corrects semantic drift at a fixed B rather than by widening the beam. HPR adds only a lightweight per-layer dual-tower scoring over a pool of N candidates, with the context-side projection computed once per request; DPD runs its two decoder towers in parallel on a shared encoder while keeping the per-channel beam width and the final Top- K list length unchanged. The combined parameter count remains below the TIGER baseline (Table 11).

HiGR vs. OneRec-style decoding (§4.3). For a slate of M items, D SIDs per item, hidden size d , beam width B , l_{slate} slate-planner layers (equal to the lazy-decoder depth of the OneRec-style baseline), and l_{item} item-generator layers, decomposing full-sequence beam decoding into greedy slate planning followed by local per-item beam search reduces the decoding complexity from

$$\mathcal{O}(B \cdot M^3 D^3 \cdot l_{\text{slate}} \cdot d) \quad (\text{OneRec-style}) \quad \longrightarrow \quad \mathcal{O}(M^3 \cdot l_{\text{slate}} \cdot d + B \cdot M D^3 \cdot l_{\text{item}} \cdot d) \quad (\text{HiGR}),$$

which underlies the layer-allocation and serving results of §4.3.4.

Table 18 | Notation. The top block lists stack-level symbols; the lower blocks list the principal chapter-specific symbols, following the corresponding source papers.

Symbol	Meaning
<i>Stack-level (used throughout)</i>	
$u, \mathcal{U}$	User; set of users
$v, \mathcal{V}$	Item; set of items
S^u	Chronological interaction history of user u
O^u	Output slate of M items for user u
M	Number of items in a slate
D	Number of SID levels per item
s^d, s_m^d	d -th SID; d -th SID of item m
B	Beam width
d_{model}	Hidden dimension of the unified encoder and the HiGR decoders
C	User / context representation (encoder output)
$\mathcal{F}_\theta$	Generative model decoding the output slate
<i>TGR-GenRank / CCFormer (§3)</i>	
$\mathbf{U}, \mathbf{S}, \mathbf{T}$	User-profile, behavior-sequence, and target-item token fields
L_u, L_s, L_t	Numbers of tokens in the three fields
L	Number of stacked CCFormer interaction blocks
m, n	Sequence and channel group sizes in subspace token mixing
k, s	Kernel size and stride of convolutional sequence compression
α, β, γ	Learnable parameters of the temporal decay $\alpha \beta^{ t_i - t_j ^\gamma}$
<i>TGR-GenRec / BARGE (§4.2)</i>	
L	Per-item SID depth; c_l is the codeword at layer l
$\mathbf{h}_0$	Pre-generation decoder hidden state (HPR context anchor)
$\mathbf{p}^{(l)}$	Cumulative path embedding $\sum_{j=1}^l \mathbf{e}_{c_j}$
$\lambda; N$	HPR reranking weight; HPR scoring-pool size
R	Householder-parameterized orthogonal rotation of OSQ-VAE
K	Length of the final merged recommendation list
<i>TGR-GenRec / HiGR (§4.3)</i>	
$z, \hat{z}$	Latent embedding and aggregated quantized form (PCRQ-VAE)
e_a^d	Codeword embedding of item a at quantization layer d
η	Commitment strength in the global-quantization loss
λ_1, λ_2	Weights of the global-quantization and contrastive losses
$\tau; w_d$	InfoNCE temperature; per-layer weight in the prefix contrastive loss
$\hat{i}_m$	Planned preference embedding for slate position m (HSD)
$l_{\text{slate}}, l_{\text{item}}$	Layer counts of the slate planner and item generator
y^+, y^-	Positive / negative slate in an ORPO preference pair
$\alpha; \beta$	ORPO alignment coefficient; ILD regularization weight
<i>TGR-Reason (§5)</i>	
K	Latent reasoning steps of LatentRec (K_{train} in training; $K=0$ serving)
K_r	Number of exported reason tokens per user
$\mathcal{R}_u$	Reason-token set of user u ; each $\mathbf{r}_{u,k}$ is a complete L -level SID
W	Forward window of Group Memory retrieval
M	Successors retained per reason-token query in GMR
$\mathbf{H}_u^p, \mathbf{H}_u^g$	Personal-Memory and Group-Memory representations

Table 19 | Default configurations and hyperparameters. CCFormer: industrial setting of [22]; BARGE: public-benchmark setting of [23] (the deployed retrieval channel is described in §4.2.5); HiGR: industrial setting of [24]; TGR-Reason: deployed configuration of §5.

Parameter	Default value
<i>CCFormer</i> (§3)	
Feature-token dimensionality d	256
Interaction blocks L	8
Subspace group sizes (m, n)	(8, 16)
Compression kernel / stride (k, s)	(3, 2)
Optimizer / learning rate / batch	Adam / 10^{-4} / 4096
Behavior-sequence length	1000 industrial; 200 public; 500–2000 in scaling
Training hardware	16× NVIDIA H20
Precision and sparse compression	BF16/FP16; INT8 (~70%) + double hashing (~50%)
Hyperparameter search	TPE, applied to every compared method
<i>BARGE</i> (§4.2)	
SID depth L / codebook sizes	4 / (512, 256, 128, 64), all layers learned
Tokenizer	OSQ-VAE: Householder rotation R , two channels
Encoder / decoder towers	2 layers, 4 heads each (shared encoder, dual decoders)
Embedding / attention / FFN dims	128 / 512 / 1024
Optimizer / batch / epochs	Adam with warmup / 256 / 200 (early stopping; 3 seeds)
Beam width B / HPR pool N	20 / 400
HPR fusion weight λ	0.25
OR-fusion operator	LSE (log-sum-exp soft OR)
Training hardware	2× NVIDIA H20
<i>HiGR</i> (§4.3)	
Hidden / FFN dimensions	$d_{\text{model}} = 512$ / $d_{\text{FFN}} = 2048$
SID depth D / codebook size	3 / 1024 per layer
Planner / generator depth	$l_{\text{slate}} = 14$ / $l_{\text{item}} = 2$
PCRQ-VAE weights	$\eta = 0.1$, $\lambda_1 = 0.1$, $\lambda_2 = 0.01$, $(w_1, w_2) = (1, 0.1)$
ORPO coefficient α	0.1
Training data	1B pre-training samples; 3% for ORPO post-training
Model scale	25M / 100M in deployed studies; 0.05B–2B in scaling
Hardware and serving	NVIDIA H20 training; L20 serving with KV caching
<i>TGR-Reason</i> (§5)	
Think backbone	Qwen3-1.7B + LoRA (backbone frozen in the reasoning stage)
Latent reasoning steps	$K_{\text{train}} = 2$; $K = 0$ at offline generation
PRL weight / temperatures	$\lambda = 0.1$; $T_b = 1$, $T_s = 5$
Soft tokens N	64 (~2.5M trainable STS parameters)
Offline decoding	vLLM batched rounds on 8× A100; 2.2× cycle speedup

References

- [1] H.-T. Cheng, L. Koc, J. Harmsen, et al. Wide & Deep Learning for Recommender Systems. *DLRS Workshop*, 2016.
- [2] P. Covington, J. Adams, and E. Sargin. Deep Neural Networks for YouTube Recommendations. *RecSys*, 2016.
- [3] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He. DeepFM: A Factorization-Machine Based Neural Network for CTR Prediction. *IJCAI / arXiv:1703.04247*, 2017.
- [4] W.-C. Kang and J. McAuley. Self-Attentive Sequential Recommendation. *ICDM*, 2018.
- [5] F. Sun, J. Liu, J. Wu, et al. BERT4Rec: Sequential Recommendation with Bidirectional Encoder Representations from Transformer. *CIKM*, 2019.
- [6] J. Zhai, L. Liao, X. Liu, et al. Actions Speak Louder than Words: Trillion-Parameter Sequential Transducers for Generative Recommendations. *ICML / arXiv:2402.17152*, 2024.
- [7] B. Zhang, L. Luo, Y. Chen, et al. Wukong: Towards a Scaling Law for Large-Scale Recommendation. *arXiv:2403.02545*, 2024.
- [8] Z. Zhang, H. Pei, J. Guo, et al. OneTrans: Unified Feature Interaction and Sequence Modeling with One Transformer in Industrial Recommender. *arXiv:2510.26104*, 2025.
- [9] R. Jiang, S. Gowal, T. A. Mann, and D. J. Rezende. Beyond Greedy Ranking: Slate Optimization via List-CVAE. *ICLR*, 2019.
- [10] Z. Liu, S. Wang, X. Wang, et al. OneRec-Think: In-Text Reasoning for Generative Recommendation. *arXiv:2510.11639*, 2025.
- [11] J. Tang, S. Dai, T. Shi, J. Xu, X. Chen, W. Chen, J. Wu, and Y. Jiang. Think Before Recommend: Unleashing the Latent Reasoning Power for Sequential Recommendation (ReaRec). *arXiv:2503.22675*, 2025.
- [12] J. Achiam, S. Adler, S. Agarwal, et al. GPT-4 Technical Report. *arXiv:2303.08774*, 2023.
- [13] H. Touvron, L. Martin, K. Stone, et al. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv:2307.09288*, 2023.
- [14] DeepSeek-AI. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv:2501.12948*, 2025.
- [15] Kimi Team. Kimi K2: Open Agentic Intelligence. *arXiv:2507.20534*, 2025.
- [16] J. Deng, S. Wang, K. Cai, et al. OneRec: Unifying Retrieve and Rank with Generative Recommender and Iterative Preference Alignment. *arXiv:2502.18965*, 2025.
- [17] G. Zhou, J. Deng, J. Zhang, et al. OneRec Technical Report. *arXiv:2506.13695*, 2025.
- [18] G. Zhou, H. Hu, H. Cheng, et al. OneRec-V2 Technical Report. *arXiv:2508.20900*, 2025.
- [19] J. Zhang, Y. Li, Y. Liu, et al. GPR: Towards a Generative Pre-Trained One-Model Paradigm for Large-Scale Advertising Recommendation. *arXiv:2511.10138*, 2025.
- [20] OneRec Team. OneReason Technical Report. *arXiv:2606.06260*, 2026.
- [21] E. Botta, J. Yang, Y.-P. Hsu, et al. PinRec: Unified Generative Retrieval for Pinterest Recommender Systems. *arXiv:2504.10507*, 2025.
- [22] Y. Wang, H. Zhang, H. Hu, Y. Li, B. Wen, J. Tu, C. Zhuo, and Z. Li. CCFormer: Efficient Cross-Field Interaction and Hierarchical Sequence Compression for Industrial Recommendation at Tencent. *arXiv:2607.28070*, 2026.

- [23] J. Zeng, J. Zhu, J. Chen, Y. Li, W. Liu, C. Zhuo, and Z. Li. Bridging the Structural Gap: Adapting Autoregressive Generation for Recommendation. *arXiv:2607.21028*, 2026.
- [24] Y. Pang, Z. Liu, Y. Li, et al. HiGR: Industrial-Scale Hierarchical Generative Slate Recommendation Framework in Tencent. *CIKM / arXiv:2512.24787*, 2026.
- [25] C. Huang, H. Huang, T. Yu, K. Xie, J. Wu, S. Zhang, J. McAuley, D. Jannach, and L. Yao. A Survey of Foundation Model-Powered Recommender Systems: From Feature-Based, Generative to Agentic Paradigms. *IEEE TKDE / arXiv:2504.16420*, 2025.
- [26] G. Zhou, X. Zhu, C. Song, et al. Deep Interest Network for Click-Through Rate Prediction. *KDD*, 2018.
- [27] Q. Pi, W. Bian, G. Zhou, X. Zhu, and K. Gai. Practice on Long Sequential User Behavior Modeling for Click-Through Rate Prediction (MIMN). *KDD*, 2019.
- [28] L. Guan, J.-Q. Yang, Z. Zhao, et al. Make It Long, Keep It Fast: End-to-End 10k-Sequence Modeling at Billion Scale on Douyin Recommendation (STCA). *WWW*, 2026.
- [29] Q. Pi, G. Zhou, Y. Zhang, et al. Search-Based User Interest Modeling with Lifelong Sequential Behavior Data for Click-Through Rate Prediction (SIM). *CIKM*, 2020.
- [30] J. Chang, C. Zhang, Z. Fu, et al. TWIN: Two-Stage Interest Network for Lifelong User Behavior Modeling in CTR Prediction at Kuaishou. *KDD*, 2023.
- [31] J. Zhu, Z. Fan, X. Zhu, et al. RankMixer: Scaling Up Ranking Models in Industrial Recommenders. *CIKM*, 2025.
- [32] C. Gao, S. Li, W. Lei, et al. KuaiRec: A Fully-Observed Dataset and Insights for Evaluating Recommender Systems. *CIKM*, 2022.
- [33] S. Rajput, N. Mehta, A. Singh, et al. Recommender Systems with Generative Retrieval. *NeurIPS*, 2023.
- [34] W. Wang, H. Bao, X. Lin, et al. Learnable Item Tokenization for Generative Recommendation. *CIKM*, 2024.
- [35] Y. Yang, et al. Sparse Meets Dense: Unified Generative Recommendations with Cascaded Sparse-Dense Representations (COBRA). *arXiv:2503.02453*, 2025.
- [36] Y. Hou, et al. ActionPiece: Contextually Tokenizing Action Sequences for Generative Recommendation. *ICML*, 2025.
- [37] Y. Yu, Y. Wang, W. Ma, Z. Guo, and M. Zhang. APAO: Adaptive Prefix-Aware Optimization for Generative Recommendation. *KDD / arXiv:2603.02730*, 2026.
- [38] J. Hong, N. Lee, and J. Thorne. ORPO: Monolithic Preference Optimization without Reference Model. *EMNLP*, 2024.
- [39] R. Rafailov, A. Sharma, E. Mitchell, et al. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. *NeurIPS*, 2023.
- [40] H. Su, B. Kong, L. Cheng, C. Zhuo, and Z. Li. Do Recommenders Need to Think in Words? Latent Reasoning for Efficient LLM-Based Recommendation. *CIKM*, 2026.
- [41] J. Zhang, B. Zhang, W. Sun, et al. Slow Thinking for Sequential Recommendation (STREAM-Rec). *arXiv:2504.09627*, 2025.
- [42] C. Yi, D. Chen, G. Guo, et al. RecGPT Technical Report. *arXiv:2507.22879*, 2025.
- [43] Q. Ding, K. Course, L. Ma, et al. Bending the Scaling Law Curve in Large-Scale Recommendation Systems. *arXiv:2602.16986*, 2026.

- [44] B. Zheng, C. Yi, D. Chen, et al. RecGPT-V3 Technical Report. *arXiv:2607.15591*, 2026.
- [45] H. Wang, Y. Wu, D. Chang, et al. Self-Evolving Recommendation System: End-To-End Autonomous Model Optimization With LLM Agents. *arXiv:2602.10226*, 2026.
- [46] H. Gao, J. Geng, W. Hua, et al. A Survey of Self-Evolving Agents: What, When, How, and Where to Evolve on the Path to Artificial Super Intelligence. *arXiv:2507.21046*, 2025.