

Connected Content Retriever: Dense Graph Edge Features for Pre-Ranking at LinkedIn

Akhilesh Gupta*
LinkedIn Corporation
Mountain View, CA, USA
akgupta@linkedin.com

Sudarshan Srinivasa
Ramanujam*
LinkedIn Corporation
Mountain View, CA, USA
sramanujam@linkedin.com

Chirag Bhanuprasad Mehta*
LinkedIn Corporation
Mountain View, CA, USA
chmehta@linkedin.com

Reshma Asharaf Beena
LinkedIn Corporation
Mountain View, CA, USA
rbeena@linkedin.com

Dhritiman Das
LinkedIn Corporation
Mountain View, CA, USA
dhdas@linkedin.com

Birjodh Singh Tiwana
LinkedIn Corporation
Mountain View, CA, USA
btiwana@linkedin.com

Bhargavkumar Kanubhai Patel
LinkedIn Corporation
Mountain View, CA, USA
bhapatel@linkedin.com

Mack Lee
LinkedIn Corporation
Mountain View, CA, USA
maclee@linkedin.com

Renyi Tang
LinkedIn Corporation
Mountain View, CA, USA
rtang@linkedin.com

Abstract

In large-scale recommendation systems like the LinkedIn Feed, content generated by a member’s network (connections and follows) makes up over 70% of impressions and engagement. It is therefore essential that the pre-ranking layer forwards the best possible few hundred candidates to the ranking layer. LinkedIn’s professional knowledge graph carries engagement signals across both the first-degree network (connections and follows) and the second-degree network - posts that a 1st-degree connection reacted to, commented on, or reshared but did not author (a.k.a. stranger viral). Due to this fan-out, the resulting candidate index exceeds one billion; selection of activities from the viewer’s network narrows it down to roughly tens of thousands of activities that must be scored within a 120 ms p_{99} latency budget. We present Connected Content Retriever (CC Retriever), a pre-ranking system that scores these candidates with a full deep ranking model on GPUs at low latency. At its core is a sorted-search GPU primitive that joins dense graph-affinity features (viewer-to-author) with document-level features stored on the GPU at runtime in 5-10 ms. The shift to GPU-served scoring enabled a 50 $\times$ scale-up of the ranking model’s parameters and delivered a +2.5% lift in content time spent on the LinkedIn Feed in online experiments, significantly higher than the typical gains observed in LinkedIn Feed experiments. In this work, we describe the feature set we leverage from LinkedIn’s economic graph and the model architecture used for scoring, with a particular emphasis on the online system that scales the stack. We present the join algorithm used to attach edge features (viewer-author features that capture pairwise affinity along the social graph, e.g., interaction recency and co-engagement) to documents, how request-level features are handled, and the GPU Retrieval-as-Ranking stack that now powers multiple use cases at LinkedIn. Ablations show that removing edge features causes a 7.86% drop in recall, underscoring the importance of graph-based connection signals that content embeddings alone were not able to capture.

*These authors contributed equally to this research.

CCS Concepts

• **Computing methodologies** $\rightarrow$ **Machine learning approaches**; **Natural language processing**; • **Information systems** $\rightarrow$ **Retrieval models and ranking**.

Keywords

Recommendation Systems, Neural Retrieval, GPU Inference, Large Scale Information Retrieval

1 Introduction

LinkedIn is the world’s largest professional network, with hundreds of millions of monthly active members consuming posts from their professional graph: first-degree connections, follows, and second-degree posts propagated through engagement. Members produce on the order of hundreds of million posts per day, but distribution is fundamentally graph-mediated rather than globally popularity-ranked: majority of Feed impressions are activities authored by, or engaged with by, an entity within the viewer’s network neighborhood. Consequently, the relevance of a candidate post is jointly determined by (i) its semantic content and (ii) the viewer-to-content creator affinity along the social graph — a signal we term connection strength and treat as a first-class ranking feature, co-equal with topical interest. This contrasts with prior approaches that encode connection only as a graph-distance prior or apply it as a post-hoc reweighting of an otherwise content only ranker.

This work focuses on the pre-ranking (L1) stage of the Feed funnel, which is subject to a strict throughput-latency contract: reduce a candidate set of tens of thousand activities per request to a relevance ordered few hundreds subset, under a scoring latency budget of 120 ms p_{99} , while preserving sufficient fidelity in both the connection-strength and content-interest dimensions to avoid starving the downstream L2 ranker of discriminative signal. The central modeling challenge is therefore to jointly embed graph-affinity and content-semantic signals into a scoring function that is (a) cheap enough to evaluate over $O(10^4)$ candidates per request, and

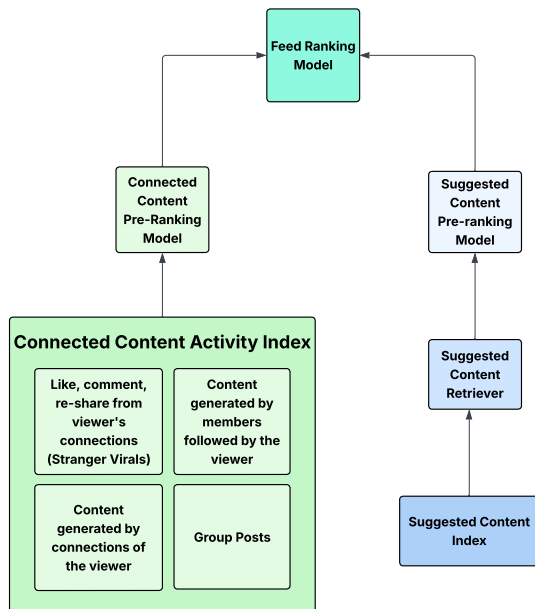


Figure 1: Feed Retrieval and Ranking System

(b) calibrated enough that its top-K cutoff is approximately recall-preserving with respect to the L2 ranker’s objective. We define an **“author”** as the LinkedIn member who originates a candidate item, whether by authoring an original post, resharing, reacting or commenting on another member’s post.

In prior work [9] we showed that LLM based causal retrieval can be effectively scaled for the unconnected content surface, where item relevance is dominated by content semantics. However, the connected-content (CC) regime is qualitatively different: pairwise viewer–author edge features (tie strength, interaction recency, co-engagement, second-degree path statistics) carry a first-order signal that cannot be recovered from content embeddings alone. We empirically observed a statistically significant degradation in online engagement metrics in A/B tests when the CC pre-ranking layer was driven by LLM content embeddings without graph-edge conditioning; we report the corresponding offline ablation in the results section to quantify the contribution of edge features to top-K recall.

The dominant constraint on this layer is its compute envelope: scoring on the order of tens of thousands of candidates per request under a P99 latency of 120 ms. Historically, this has forced the CC pre-ranker into a low-capacity regime — typically a shallow MLP or two-tower model with on the order of $O(10^5)$ trainable parameters — sized to amortize edge-feature lookup and per-candidate scoring within budget on CPU [2]. This capacity ceiling is the main bottleneck that prevents the layer from expressing higher-order interactions between the content and the graph signals.

The recent shift to GPU-resident scoring has enabled a class of retrieval-as-ranking (RAR) architectures in industrial recommender systems [1, 4, 15], in which a single model jointly performs candidate retrieval and fine-grained scoring on accelerator hardware. However, existing RAR formulations are predominantly two-tower

or late-interaction designs optimized for content-only retrieval, and do not provide a tractable mechanism for fusing high-cardinality, pair-dependent edge features into the scoring function at the latency targets required for industrial pre-ranking — fundamentally because edge features are not factorizable into a static item-side representation. In this work we present an end-to-end redesign of the CC pre-ranking stack as a GPU-RAR system that (i) materializes viewer–candidate edge features inline on the GPU, (ii) fuses them with content embeddings through a non-factorizable interaction module, and (iii) sustains the tens of thousands candidates scoring at 120 ms p_{99} contract while operating at a parameter scale several orders of magnitude beyond the prior CPU-bound baseline.

2 Related Work

LinkedIn’s LiNR [1] pioneers live-updated, model-based retrieval with billion-scale indexes embedded in a differentiable GPU. This approach, named GPU Retrieval as Ranking (GPU-RAR), loads the entire candidate set into GPU memory and scores all candidates against the query in parallel via dense matrix operations, replacing approximate nearest-neighbor index lookups with exhaustive GPU search. SilverTorch [13] extends this direction further. Instead of calling out to separate ANN and filtering services, it implements both as tensor operations: a fused int8 Inverted File Index kernel for nearest-neighbor search and a GPU bloom index that performs filtering with bitwise tensor operations. Both papers show that when retrieval runs on GPU, new opportunities emerge for richer scoring than two-tower decomposition allows.

Building on this model-based retrieval paradigm, Tencent [7] deploys Wide & Deep at retrieval scale by augmenting dual-tower embeddings with explicit sparse cross-feature interactions. These cross-features are binary indicators of whether a user has interacted with ads sharing a given attribute (categories, advertiser metadata, etc.), gated against real-valued aggregates such as click counts and CTRs over 1, 7, 15, and 30-day windows, and accelerated via a GPU-optimized compressed inverted list. While the system introduces explicit feature interaction into retrieval, the interaction module itself remains a sparse weighted aggregation operator rather than a general neural scorer operating over dynamically joined dense cross-features.

Beyond linear wide components, several systems pursue richer query-item interactions at retrieval scale through alternative architectural routes. SparCode [12] enables all-to-all feature interactions between queries and items via an expressive parameterized scorer (e.g., MLPs, attention, or CrossNet), made tractable at retrieval time by quantizing queries into a small vocabulary of discrete codes so that the scorer can be evaluated offline over all (code, item) pairs, with the resulting scores cached in a sparse inverted index that is jointly trained end-to-end with the scorer and the codebook. HSN [10] introduces user-ad interaction features and an expressive non-linear scorer into retrieval while preserving sub-linear cost by jointly learning a hierarchical clustering of items together with the scorer, so that the costly interaction computation runs once per cluster rather than once per item. Both approaches escape the inner-product constraint of two-tower models by introducing an intermediate vocabulary, either codes over queries or clusters over items, that lets expensive scoring be amortized away from

per-item online work, but they do not join explicit cross-features at serving time.

A related line of work focuses on richer query-item interactions at pre-ranking through architectural means. GESR [4] deploys a Mixture of Attention, combining hard-match attention, target-aware self-attention, and cross-attention, supported by custom kernels and serving caches. TARQ [8] brings target attention into pre-ranking at Taobao via residual quantization, with attention scores precomputed against a learned codebook. HIT [14] keeps two-tower decoupling intact and instead pregenerates holistic interaction vectors aligned at training time. Across these systems, the gains come from the modeling architectures that enable richer interactions. To fit within a millisecond-scale serving budget, techniques such as kernel-level optimization and residual quantization are employed.

Another line of work extends the cross-feature signal vocabulary available at serving time. InteractRank [5] augments a two-tower model at Pinterest with query-item statistics derived from two years of logs. AIF [6] performs user-side computation in parallel with retrieval and item-side computation in a nearline manner, with the residual user-item interaction approximated at serving time. In both cases, explicit cross-features enter pre-ranking through offline precomputation, with the resulting freshness characteristics depending on the update cadence.

3 Features for Pre-Ranking Model

We use 3 different types of features for scoring our models.

- Document Features which is an attribute of candidate post (examples include item popularity, content embeddings of the post etc.)
- Request Features which are unique to a request (examples include viewer specific features like viewer embedding).
- Edge features which need both viewer and author of each candidate post to derive the feature

Document features are indexed in GPU for scoring and request features are passed to the GPU at request time. Required attributes to construct the edge features are also passed to the GPU at request time and the edge feature is joined with the required documents.

3.1 Edge Feature Join At Scoring Time Online

Edge features capture the *relationship* between a viewer and content creator (author), independent of the specific post.

Viewer-Author Action Affinity is an example of edge feature which encodes the viewer’s historical engagement with each author across action types (like, comment, share, click, long-dwell, follow, etc.) and time buckets (last hour, last day, last week, etc.). This multi-dimensional vector captures both the intensity and the recency of engagement. Edge features like these are highly predictive since they provide strong information on the connection strength between the viewer and the author. However, joining them to the required documents at request time is computationally expensive given our latency cost.

At serving time, we receive N candidate documents, each with an author ID: $\mathbf{d} = [d_1, d_2, \dots, d_N]$ and K viewer-author affinity records $(\mathbf{a}, \mathbf{F})$ where $\mathbf{a} = [a_1, \dots, a_K]$ are author IDs and $\mathbf{F} \in \mathbb{R}^{K \times D}$ are the corresponding feature vectors. We need to produce output

Algorithm 1 Edge Feature Join via Binary Search

Require: Document author IDs $\mathbf{d} \in \mathbb{Z}^{B \times N}$

Require: Affinity features $\mathbf{F} \in \mathbb{R}^{B \times K \times D}$

Require: Affinity author IDs $\mathbf{a} \in \mathbb{Z}^{B \times K}$

Ensure: Output features $\mathbf{O} \in \mathbb{R}^{B \times N \times D}$

```

1: // Step 1: Sort affinity records by author ID
2:  $\mathbf{a}_{sorted}, \mathbf{idx} \leftarrow \text{sort}(\mathbf{a}, \text{dim} = 1)$   $O(K \log K)$ 
3:  $\mathbf{F}_{sorted} \leftarrow \text{gather}(\mathbf{F}, \mathbf{idx})$ 
4: // Step 2: Binary search for each document
5:  $\mathbf{pos} \leftarrow \text{searchsorted}(\mathbf{a}_{sorted}, \mathbf{d})$   $O(N \log K)$ 
6: // Step 3: Validate matches (clamp index to avoid OOB)
7:  $\mathbf{pos}_{safe} \leftarrow \min(\mathbf{pos}, K-1)$ 
8:  $\mathbf{valid} \leftarrow (\mathbf{pos} < K) \wedge (\mathbf{a}_{sorted}[\mathbf{pos}_{safe}] = \mathbf{d})$ 
9: // Step 4: Gather features for matches
10:  $\mathbf{O} \leftarrow \text{Zeros}(B \times N \times D)$ 
11:  $\mathbf{O}[\mathbf{valid}] \leftarrow \mathbf{F}_{sorted}[\mathbf{pos}_{safe}[\mathbf{valid}]]$ 
12: return  $\mathbf{O}$ 

```

features $\mathbf{O} \in \mathbb{R}^{N \times D}$ where:

$$O_j = \begin{cases} F_i & \text{if } \exists i : a_i = d_j \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The naive approach would scan all K affinity records for each candidate’s author ID d_j to find a match, yielding $O(N \cdot K)$ operations. With N in the tens of thousands of candidate documents per request and K on the order of a few thousand viewer-author affinity records, this product is in the order of 10^7 which is prohibitive within our pre-ranking latency target.

Algorithm 1 leverages the fact that we can sort the viewer’s affinity records once and then use binary search to find matches for all documents efficiently.

Complexity analysis:

- Sort: $O(K \log K)$
- Binary search: $O(N \log K)$
- **Total:** $O(K \log K + N \log K) = O((N + K) \log K)$, dominated by $N \log K$ when $N \gg K$ which is up to $200\times$ fewer operations than the $O(N \cdot K)$ naive scan

4 Modeling Architecture

This section describes the Connected Content (CC) pre-ranking model. Given the three feature classes introduced in Section 3 - request features x_v , document features x_d , and edge features x_e , the model emits logits for three engagement objectives (click, total-Contributions (union of likes, comments and shares), long-dwell). Finally we use tunable weights (hyper-parameters) to compute the final score of the retriever model.

4.1 Architecture

The CC pre-ranking model $f_\theta(x_v, x_d, x_e)$ is a multi-task feed-forward network with $\sim O(10^6)$ parameters, designed to score tens of thousands of candidates per request inside a 120 ms p99 latency budget. The architecture is shown in Figure 2 the subsections below detail each block. Document features and edge features were already discussed in section 3 and we will not describe these again.

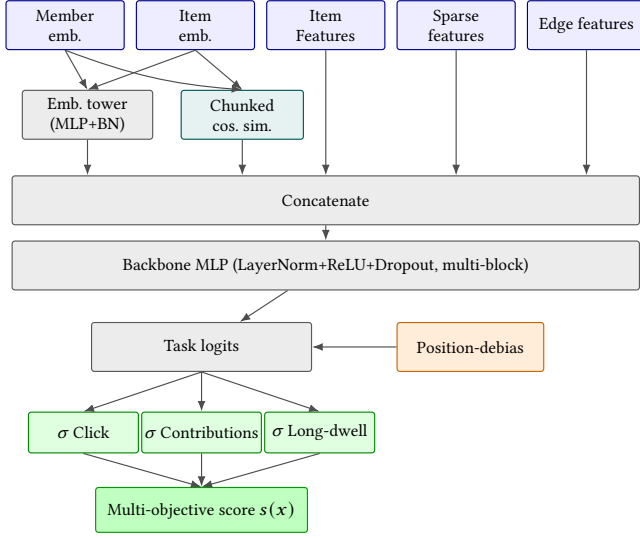


Figure 2: CC pre-ranking model architecture. Inputs are pretrained member and item embeddings, log-normalized count features, multi-hot sparse features, and edge features x_e joined as dense tensors per Algorithm 1. Pretrained embeddings pass through a ReLU + batch-norm tower; a chunked cosine-similarity block is computed in the forward pass. All signals are concatenated and consumed by a backbone MLP whose logits are additively corrected by a Feed-position-debias embedding. Sigmoid task heads for click, total contributions and long-dwell are combined via tunable weights

4.1.1 Embedding Tower. The Pre-trained member (150-d from the top 3 medoids) and item (50-d) embeddings [11] are concatenated into a 200-d input and passed through a 2-layer ReLU + batch-norm sub-network. Batch-norm is used here so that running statistics provide a stable input scale to the downstream backbone.

4.1.2 Derived Features. The member representation in [11] is the concatenation of three 50-dimensional medoid embeddings; we treat each medoid as one 50-dim chunk aligned with the 50-d item embedding and compute a cosine similarity per chunk in float32, resulting in three scalars which are converted back to float16.

Similarly we convert large integer count features to rate based features and apply log transformation bringing all of these to float16.

4.1.3 Request Features. Feature values for viewer specific features are passed along with the request. For the viewer side embeddings apart from the medoid based embeddings passed in the embedding tower, we also make use of the outputs of the final transformer layer of our sequence based ranking model which encodes information on all historical engagement of the member. [3]

4.1.4 Backbone MLP. The fused feature vector $x \in \mathbb{R}^{d_{in}}$ flows through a stack of identical blocks of the form

$$h_{\ell+1} = \text{Dropout}(\text{LayerNorm}(\text{ReLU}(W_{\ell}h_{\ell} + b_{\ell}))), \quad (2)$$

followed by a final unactivated linear head producing the three task logits. Linear weights are Kaiming-normal initialized, biases are zero, and a small dropout rate is applied inside each hidden block.

4.1.5 Position-Debias Head. To prevent the backbone from learning ranker position bias, the Feed slot is treated as a *debiasing covariate* rather than a feature: a learned embedding table E_{pos} keyed by the (clipped) Feed slot is added to the backbone logits [16],

$$\hat{y} = \text{Backbone}(x) + E_{\text{pos}}(\min(\text{slot}, S_{\text{max}})), \quad (3)$$

where S_{max} is a cutoff above which slots are bucketed together. During training, a moderate dropout is applied to the position contribution; at inference and during validation the slot is clamped to a constant, zeroing out the bias term and yielding relevance-only logits.

4.1.6 Multi-Objective Score. At serving time the model emits logits for three engagement objectives:

- **click**
- **totalContributions** — union of likes, comments and shares
- **long-dwell**

The three predicted probabilities are combined via a linear combination with tunable per-objective weights (hyper-parameters) to produce the final retriever score.

$$s(x) = \sum_{k \in \{\text{click}, \text{totalContributions}, \text{long-dwell}\}} w_k \sigma(\hat{y}_k(x)). \quad (4)$$

We also add an additional freshness boost to the final score based on age to encourage exploration in the system.

5 Offline Evaluation

We use recall@k for offline evaluation to check how well the pre-ranking model is aligned with the ranking model. The final ranked order of each session is taken as oracle. Candidates to the same session are scored with the retriever and we evaluate recall@10 to compare different variants of the offline retriever model. We chose 10 since it was most aligned with online metrics based on our empirical studies.

$$\text{recall}@k = \frac{|R_K \cap M_k|}{K},$$

where R_K is the ranker’s top- K activities (the relevant set) and M_k is the retriever’s top- k activities.

6 Online System

6.1 End-to-End System

This section describes the end-to-end online serving architecture at scale. The system scores tens of thousands of candidate documents per request using a learned model with dense embeddings, document features, and user-document edge features, all within the latency budget of a retrieval stage (~ 120 ms). We present a general architecture that decouples three concerns: (1) candidate selection from a large corpus, (2) feature storage and retrieval at corpus scale, and (3) GPU-accelerated model scoring with batched inference. This separation enables independent scaling of each component and is

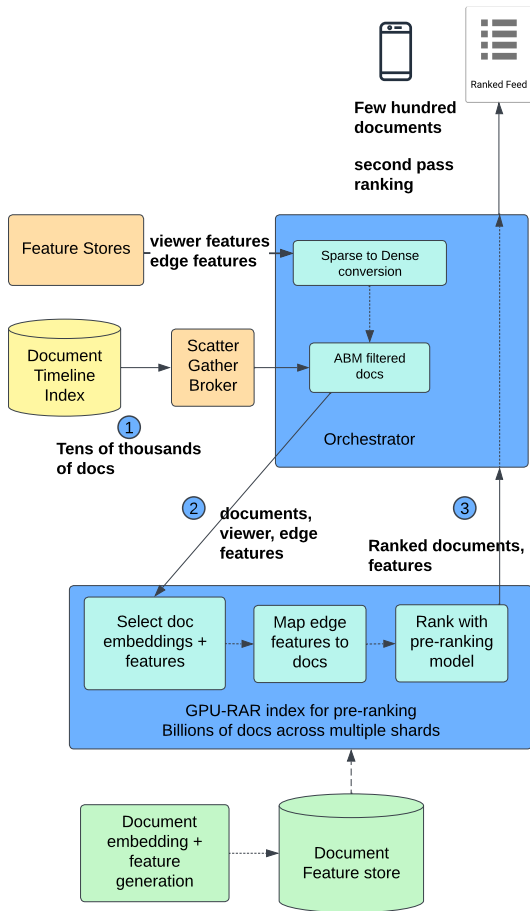


Figure 3: End-to-end architecture of the retrieval-as-ranking system. A user request flows through three stages: (1) candidate selection from a graph-based index filters the corpus from billions to tens of thousand documents, (2) the retrieval service generates request-level and edge features and assembles the scoring request, and (3) the GPU inference server resolves candidate IDs to pre-indexed feature tensors, executes a batched model forward pass, and returns top- K results to the ranking model.

applicable to any recommendation system where the retrieval stage must go beyond embedding similarity to incorporate rich features.

6.1.1 System Overview. The online serving pipeline consists of three cooperating subsystems that execute in sequence for each user request.

6.1.2 Stage 1: Candidate Selection via Graph-Based Filtering. The first stage reduces the full document corpus (N_{corpus}) to a tractable candidate set ($N_{\text{candidates}}$) for model scoring. In our Feed recommendation scenario, the relevant document set for a user is defined by a social graph: only documents authored or interacted with by entities connected to the user are eligible. We use a **timeline**

index that maintains the mapping from document authors to the timeline of their recent documents. This index supports attribute-based matching (ABM): given a user, it returns all document IDs whose author is within the user’s network. This filtering operation reduces the corpus by 4–5 orders of magnitude (from billions to tens of thousands).

6.1.2.1 Key design decisions. The timeline index uses a **scatter-gather** pattern across shards. The index is partitioned (e.g., by document author ID), and an orchestration broker fans out the user’s network membership to the relevant shards, collects matching document IDs, and deduplicates. This allows the index to scale horizontally with corpus size.

6.1.3 Stage 2: Retrieval Service (Orchestration & Feature Assembly). The retrieval service is the orchestrator between candidate selection and GPU scoring. It receives the filtered candidate IDs from Stage 1, computes user-level and request-level features once per request, looks up the user-keyed edge affinity tensors (Section 3), and assembles a compact scoring payload comprising the candidate IDs, the user’s query embedding, request-level features, and the edge affinity tensors keyed by entity ID. Crucially, **document features and document embeddings are not transmitted in the payload**; they are pre-indexed in the GPU inference server (Section 6.1.4) and resolved server-side by document ID, reducing per-request payload size by orders of magnitude.

6.1.4 Stage 3: GPU Inference Server. The GPU inference server receives candidate IDs and user-side features from Stage 2, resolves them to pre-indexed feature tensors, and executes a batched model forward pass to produce the final ranking. Its architecture, including corpus-scale ingestion, ID resolution, batched scoring, and memory management is described in detail in Sections 6.2 and 6.3.

6.2 Ingestion Pipeline: Keeping the Corpus Index Fresh

To keep the GPU index fresh against continuous document creation, update, and aging, we follow LiNR’s offline-batch-plus-nearline-CDC ingestion pattern [1], consuming document updates as a stream and writing them to the GPU index in near-real-time. Our pipeline extends this with a two-class feature split: static features computed once at document creation (e.g., embeddings, content type), and dynamic features such as `activityPopularity` that evolve with member engagement. The two classes flow through separate streams so the index can refresh them independently rather than reprocessing the full feature set on every change. Static features arrive on an availability event triggered by document creation; any fields missing from that event are hydrated via gRPC calls to the corresponding feature services. Dynamic features are produced by a stateful streaming job that aggregates member interactions over fixed windows and emits periodic updates on its own stream. The ingestion job consumes both streams in parallel and writes updates independently to the index.

6.2.0.1 Storage efficiency at scale. At billions of documents, per-record layout choices have outsized impact on GPU memory. We extend LiNR’s compact-integer attribute encoding [1] to our categorical features using 32-bit integers, enabling vectorized GPU

comparison during filtering and yielding an order-of-magnitude reduction over string representations. A daily background job evicts documents past the retention window, bounding the active corpus.

6.2.0.2 Scaling ingestion for a billion-document corpus. The ingestion subsystem maintains a GPU-resident index while processing up to tens of thousands of updates per second — orders of magnitude above both the platform’s post creation rate and LiNR’s reported update rates [1] — with a maximum staleness target of a couple of minutes. Each post induces a long tail of downstream update events (engagement signals, attribute changes, reshares) that the index must continuously absorb to stay fresh. A CPU-resident parallel hash map $H : \text{DocID} \rightarrow \text{Position}$ (tens of GB for billions of entries) maps document IDs to their GPU tensor positions. Ingestion operates in two modes:

6.2.0.3 Bootstrap (server cold start). Bootstrap replays the CDC stream at cold start [1]; we batch documents in groups of hundreds of thousands and apply a four-step per-batch protocol: (1) parallel-identify new vs. existing document IDs via parallel iterator over H , (2) sequential prefix-sum to assign storage positions for new documents, (3) GPU scatter features to assigned positions, and (4) update the hash map only after successful writes for transactional consistency. An earlier implementation used a tensor membership check (`isin`) for batch lookups; replacing it with the parallel hash-map iterator gave a $52\times$ speedup in average batch insert time.

6.2.0.4 Steady-state (serving + ingestion). Once serving begins, ingestion batch sizes are reduced by $3\times$ to avoid contending with model execution for GPU memory bandwidth. Parallel iterator performance keeps ingestion lag under 1 minute.

6.3 GPU Retrieval as Ranking Beyond Embedding Features

The GPU Retrieval-as-Ranking server is a Rust-based inference engine that maintains a real-time document index entirely in GPU memory. The Feed retrieval engine sends tens of thousands candidate (p_{99}) IDs per request; the inference server resolves them to feature tensors, assembles batched inputs, and executes a TorchScript model. The corpus of billions of documents are partitioned into multiple shards (S) (few hundred million docs in each). Each document stores a set of feature tensors $\{f_1, f_2, \dots, f_k\}$ —such as embedding vectors, activity signals, and content type encodings—with a combined per-document footprint of $D = \sum_{i=1}^k \text{sizeof}(f_i)$ bytes. Each shard must satisfy the memory constraint:

$$\frac{N}{S} \times D \leq M_{\text{GPU}} - M_{\text{headroom}} \quad (5)$$

where M_{GPU} is the total device memory and M_{headroom} reserves capacity for model weights, intermediate activations, and CUDA allocator overhead.

6.3.1 Scaling Inference: Reducing GPU Cost at Production Throughput.

Achieving few thousand QPS at $p_{99} \leq 120\text{ms}$ with a billions of documents GPU-resident index required minimizing GPU idle time and maximizing utilization per device. The fewer GPU cycles spent on non-inference work, the fewer shards and GPUs needed

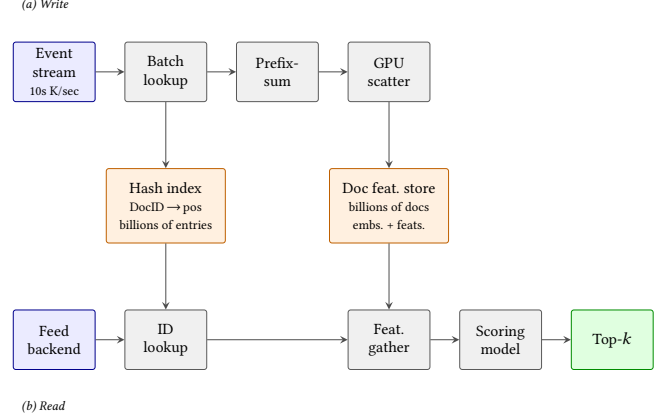


Figure 4: Architecture of the GPU inference server. The write path (top) ingests events through a batch lookup, prefix-sum offsets, and a GPU scatter that updates the shared hash index and document feature store. The read path (bottom) resolves candidate document IDs against the hash index on the CPU, gathers features on the GPU, runs a batched scoring-model forward pass, and selects the top- k documents.

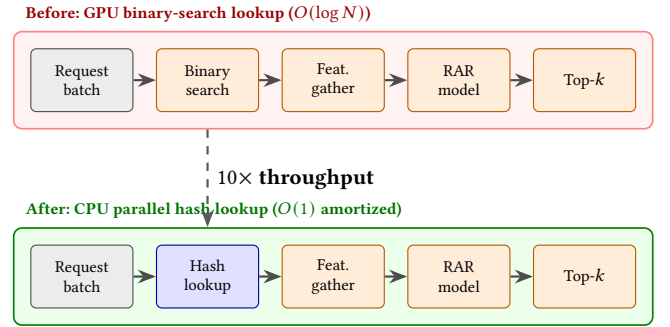


Figure 5: Effect of moving document ID-to-position resolution from GPU-side binary search to CPU-side parallel hash lookups. GPU binary search costs $O(\log N)$ per lookup and its latency scales with batch size; CPU parallel hash lookups are $O(1)$ amortized and stay roughly constant with batch size, yielding a $10\times$ throughput improvement on the inference path. Orange boxes are GPU operations, blue boxes CPU operations.

to meet throughput targets. We achieved this through three key optimizations:

- moving document ID resolution off the GPU
- batching variable-length requests into fixed-size tensors
- tuning the memory allocator to prevent fragmentation under sustained load

6.3.1.1 Document ID Resolution: GPU Kernel to CPU Parallel Lookup.

Each request carries tens of thousands of candidate IDs pre-filtered by upstream retrieval. The GPU inference server must resolve these to index positions using a hash map H that maps each document ID to its storage location in the GPU index: for every candidate q_i , the server computes

$$\text{pos}(q_i) = \begin{cases} H[q_i] & \text{if } q_i \in H \\ -1 & \text{otherwise} \end{cases} \quad (6)$$

This resolution happens under a read lock on the index — a read-write lock at the index granularity. Each in-flight batch holds the read lock, while ingestion takes an exclusive write lock per upsert batch.

Resolving candidate IDs to index positions is structurally a hash-lookup problem, but a GPU-resident hash map is impractical for a streaming-update index: the ingestion rate (tens of thousands of upserts/sec, Section 6.2) would force frequent rebuilds, and the irregular, pointer-chasing access pattern is a poor match for GPU coalesced-memory execution. We therefore started with the standard workaround for keyed lookup on the GPU: store the ID→position mapping as a sorted array in HBM and resolve each query with a custom binary-search kernel. For a single request, this completed in ~5ms over a few-hundred-million-entry index. However, once we added batching to improve GPU utilization, the kernel had to search $b \times$ tens of thousands of IDs against the same index, and its $O(\log N)$ per-lookup cost compounded with sequential HBM access. Profiling showed the kernel consumed >50% of model latency and scaled linearly with batch size (~5ms at batch 1 → ~47ms at batch 8), capping effective QPS.

Our solution was to move ID resolution off the GPU entirely, splitting work along the natural hardware boundary: irregular pointer-chasing on the CPU, dense tensor ops on the GPU. A CPU-resident parallel hash map resolves each ID in $O(1)$ amortized time and processes the full $b \times$ tens-of-thousands batch concurrently across CPU threads in ~1ms, independent of batch size. This freed the GPU exclusively for feature gathering and model execution, yielding a $10\times$ throughput improvement. We also evaluated GPU-resident hash map implementations for $O(1)$ on-device lookups, but at our ingestion rate of tens of thousands of upserts/sec the hash map would require continuous rebuilds, negating its throughput advantage; for workloads with a near-static index this trade-off would flip.

6.3.1.2 Batching and Padding for Variable-Length Requests.

Requests carry variable candidate counts after position resolution. For a batch of b requests with n_i valid candidates each, we pad to $n_{\max} = \max(n_i)$ using a scatter-based approach:

- (1) Compute scatter indices (batch_idx, pos_idx) from valid position masks—done once and reused for all features.
- (2) Flatten valid positions across the batch and single-gather features from the GPU index.
- (3) Scatter into the output tensor: $T[\text{batch_idx}, :, \text{pos_idx}] = \text{gathered_features}$.

Example with $b = 2$: R_0 has valid positions [10, 42, 87], R_1 has [5, 63].

- (1) Scatter indices: $R_0 \rightarrow (0, 0), (0, 1), (0, 2)$; $R_1 \rightarrow (1, 0), (1, 1)$

- (2) Flatten positions $\rightarrow [10, 42, 87, 5, 63]$, single gather from GPU index
- (3) Scatter into $T[b=2, n_{\max}=3, \text{features}]$:

$$\begin{aligned} T[0, :] &= [\text{features}[10], \text{features}[42], \text{features}[87]] \\ T[1, :] &= [\text{features}[5], \text{features}[63], \text{pad}] \end{aligned}$$

Padded positions can produce non-trivial MLP scores, so we mask them out with scores[i, j] = $-\infty$.

6.3.1.3 GPU Memory Tuning Under Constrained Headroom.

With few hundred million document tensors consuming the majority of GPU memory per shard, the remaining headroom for model activations is tightly constrained:

$$M_{\text{total}} = M_{\text{index}} + M_{\text{model}} + M_{\text{activation}} + M_{\text{allocator}} \quad (7)$$

During serving, each batch requires the model to score $b \times n_{\max}$ candidates, where b is the batch size and $n_{\max}$ is the count of candidates padded per request. For a model with few million parameters, each forward pass allocates intermediate activation tensors proportional to $b \times n_{\max} \times d_{\text{hidden}}$ across both layers, plus temporary tensors for feature gathering, embedding dot products, and affinity lookups. Batching and high volume of documents to score creates significant transient memory pressure on top of the document index already occupying the majority of GPU memory.

Because the document index occupies the majority of HBM per shard, only a few hundred MB of contiguous headroom remain for the forward pass. Under variable-batch load, this headroom fragmented and produced two distinct failures: (i) classic fragmentation OOM, and (ii) CUBLAS_STATUS_ALLOC_FAILED, where cuBLAS could not claim a workspace from the fragmented headroom mid-forward. We address them independently. Setting the PyTorch garbage-collection threshold to 0.6 reclaims unused cached blocks once 60% of reserved memory is occupied, bounding fragmentation across batch sizes. We then pre-allocate a fixed cuBLAS workspace pool of 10×4096 KB (~40 MB): 4096 KB is the smallest per-workspace size at which cuBLAS still selects the optimal Hopper matmul kernel, and depth 10 matches in-flight CUDA stream concurrency at p_{99} , eliminating per-stream workspace contention.

7 Results

7.1 GPU-RAR Benchmarking Results

Table 1 shows the effect of each optimization on the QPS handled. Moving the lookups to the CPU had the biggest effect.

Table 1: Throughput progression of the GPU inference server under a 120ms p_{99} SLA. Each row adds one optimization to the previous configuration. Batch size denotes concurrent requests scored in a single GPU forward pass.

Stage	Configuration	Batch Size	QPS
0	Baseline: single-request MLP forward	b	OOM
1	+ memory tuning + pre-materialized tensors	b	x
2	+ multi-request batching	$\leq 16b$	1.75x
3	+ CPU parallel hash map lookups	32b	10x

7.2 Modeling Results

The following section illustrates the AB test results when a trained model with all the features and infra optimizations discussed were ramped online against the older model and system.

- Content Time Spent in LinkedIn up by **2.5%**
- Increase in volume of sessions by **0.1%**
- Reduction in skipped updates by **0.8%**
- Increase in professional interaction in the platform (active and passive engagement combined) by **2.68%**

All results cited above have p-values < 0.0001 and are statistically significant.

Table 2: Feature ablation on the CC pre-ranker. Each row disables one input stream while holding architecture and training identical to the full model. Recall@10 is reported as a percent change relative to the full model on a fixed offline evaluation dataset.

Variant	Recall@10 Δ
Full model (this work)	—
w/o edge features	−7.86%
w/o content features	−3.05%
w/o item features	−2.5%

7.2.1 Ablation Studies on Features. Table 2 shows the impact that various features have on the recall metric. We can clearly see that the edge features have the heaviest impact when removed in the model on top of LLM based content features and other item features.

7.2.2 Parameter-Scaling Ablation. To verify that the deployed few-million parameter backbone is the right operating point, we ran a controlled scaling sweep that holds the feature set, training recipe, and embedding tower fixed and co-scales backbone width and depth across two orders of magnitude in parameter count. Offline retrieval quality saturates quickly past the anchor: scaling nearly two orders of magnitude beyond the deployed size yields only +0.20% Recall@10, with marginal returns falling below 0.01% per doubling at the upper end (Table 3). The shape matches diminishing-returns scaling laws reported for tabular and CTR models.

Online throughput pays a steep and roughly linear cost for those negligible offline gains. Under a 120 ms p_{99} SLA, each doubling of the backbone costs roughly 10–15% of sustainable per-replica QPS through the mid-range; near two orders of magnitude beyond the deployed size, sustainable QPS halves; beyond that, throughput collapses further, and the largest variant we attempted fails to load on a single GPU due to memory pressure. The deployed operating point therefore was kept at $\sim O(10^6)$ for this tradeoff.

8 Conclusion

In this work we showed how we completely overhauled the connected content retrieval and pre-ranking stack in LinkedIn. We were able to move to GPU based pre-ranking where we could score larger models with complex edge based features for a large document set under 120 ms p_{99} latency. We shared the end to end infrastructure and all the optimizations done to ship this system online which

Table 3: Joint parameter-scaling tradeoff. Backbones are reported as parameter-count multipliers of the deployed anchor (1×). Δ Recall@10 is relative to the anchor; relative QPS is per-replica throughput at a 120 ms p_{99} SLA, normalized to the smallest measured variant (1.00×). Offline quality saturates quickly while throughput degrades nearly linearly with parameter count.

Backbone (vs. deployed)	Δ Recall@10	Rel. QPS
Deployed (1×)	—	deployed
$\sim 4\times$	+0.11%	0.93×
$\sim 8\times$	+0.15%	0.89×
$\sim 15\times$	+0.16%	0.78×
$\sim 40\times$	+0.20%	0.67×
$\sim 75\times$	+0.20%	0.56×
$\sim 400\times$	N/A	OOM

is now powering LinkedIn Feed. We have only just begun in this domain. Following this ground up rebuild, we intend to try more complex modeling architectures at the pre-ranking layer and are looking into sequence modeling. In parallel we are also working on leveraging more powerful language models to encode topical relevance along with the strong graph based features from the LinkedIn economic graph to power the Feed.

Acknowledgements

This work represents the joint efforts across multiple teams in LinkedIn without whom this would not have been possible. We would like to thank (in alphabetical order) Adam Tang, Aman Mehrotra, Amithesh Ramesh, Amy Yang, Andrei Akterskii, Angela Shao, Anita Savadi, Antonio Alonso, Aslan (Mu) Bai, Bef Ayenew, Bo Wang, Christine Choi, Deepak Manoharan, Dru Pollini, Faizaan Charania, Fenil Fadadu, Hemeng Tao, Hristo Danchev, Jeffrey Zhao, Manika Agarwal, Mehdi Dabbagh, Mina Doroud, Mohit Kothari, Nick Ghotbi, Parminder Gill, Ping Jin, Pratik Dixit, Rahul Raja, Rajdeep Das, Rajeev Kumar, Rakesh Chalasani, Ronak Kaoshik, Russell Chan, Samaneh Moghaddam, Satyajeet Kumar, Saurabh Kataria, Shirley Jiang, Siddharth Dangi, Steve Jan, Swapnil Rajaram Patil, Tim Chao, Tim Jurka, Tugrul Bingol, Vibhuti Sengar, Vishal Shah, Wei Zhou, Yi Shen, Ying Xuan, Youngchae Kim, Yubo Ouyang, and Yuyang Tian for contributing to and supporting this work.

In preparing this paper, we used Claude Opus 4.6 (Anthropic) to identify and correct typos, grammatical errors, and collaboratively generate flow diagrams presented in the paper. Algorithm 1 was also generated by Claude to convert production code to pseudocode. We acknowledge the contributions of Claude in enhancing the writing process while maintaining full academic and professional integrity.

No AI tools were used for experimental design, model training, hyperparameter selection, statistical analysis, A/B test execution, or the formulation of scientific conclusions. All offline metrics, online A/B test results, and throughput benchmarks reported in this paper were produced by the authors using LinkedIn’s production infrastructure. The authors reviewed, validated, and approved all AI-generated content and retain full responsibility for the scientific accuracy and integrity of this work.

References

- [1] Fedor Borisjuk, Qingquan Song, Mingzhou Zhou, Ganesh Parameswaran, Madhu Arun, Siva Popuri, Tugrul Bingol, Zhuotao Pei, Kuang-Hsuan Lee, Lu Zheng, Qizhan Shao, Ali Naqvi, Sen Zhou, and Aman Gupta. 2024. LiNR: Model Based Neural Retrieval on GPUs at LinkedIn. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management (CIKM '24)*. ACM, New York, NY, USA, 1–8. doi:10.1145/3627673.3680091
- [2] Swapnil Ghike and Shubham Gupta. 2016. FollowFeed: LinkedIn's Feed Made Faster and Smarter. LinkedIn Engineering Blog. <https://www.linkedin.com/blog/engineering/feed/followfeed-linkedin-s-feed-made-faster-and-smarter> Accessed: 2026-04-30.
- [3] Lars Hertel, Gaurav Srivastava, Syed Ali Naqvi, Satyam Kumar, Yue Zhang, Borja Ocejó, Benjamin Zelditch, Adrian Enghardt, Hailing Cheng, Andy Hu, et al. 2026. An Industrial-Scale Sequential Recommender for LinkedIn Feed Ranking. *arXiv preprint arXiv:2602.12354* (2026).
- [4] Juhee Hong, Meng Liu, Shengzhi Wang, Xiaoheng Mao, Huihui Cheng, Leon Gao, Christopher Leung, Jin Zhou, Chandra Mouli Sekar, Zhao Zhu, Ruochen Liu, Tuan Trieu, Dawei Sun, Jeet Kanjani, Rui Li, Jing Qian, Xuan Cao, Minjie Fan, and Mingze Gao. 2025. Generative Early Stage Ranking. arXiv:2511.21095
- [5] Sujay Khandagale, Bhawna Juneja, Prabhat Agarwal, Aditya Subramanian, Jaewon Yang, and Yuting Wang. 2025. InteractRank: Personalized Web-Scale Search Pre-Ranking with Cross Interaction Features. In *Companion Proceedings of the ACM Web Conference 2025 (WWW Companion '25)*. ACM, New York, NY, USA, 1–9. doi:10.1145/3701716.3715239
- [6] Zhi Kou, Xiang-Rong Sheng, Shuguang Han, Zhishan Zhao, Yueyao Cheng, Han Zhu, Jian Xu, and Bo Zheng. 2025. AIF: Asynchronous Inference Framework for Cost-Effective Pre-Ranking. arXiv:2511.12934
- [7] Yifan Lei, Jiahua Luo, Tingyu Jiang, Bo Zhang, Lifeng Wang, Dapeng Liu, Zhaoren Wu, Haijie Gu, Huan Yu, and Jie Jiang. 2025. An Efficient Embedding Based Ad Retrieval with GPU-Powered Feature Interaction. arXiv:2511.22460
- [8] Yutong Li, Yu Zhu, Yichen Qiao, Ziyu Guan, Lv Shao, Tong Liu, and Bo Zheng. 2025. Equip Pre-ranking with Target Attention by Residual Quantization. arXiv:2509.16931
- [9] Sudarshan Srinivasa Ramanujam, Antonio Alonso, Saurabh Kataria, Siddharth Dangi, Akhilesh Gupta, Birjodh Singh Tiwana, Manas Haribhai Somaiya, Luke Simon, David Byrne, Sojeong Ha, Sen Zhou, Andrei Akterskii, Zhanglong Liu, Samira Sriram, Zihan Xiong, Zhoutao Pei, Angela Shao, Alex Li, Annie Xiao, Caitlin Kolb, Thomas Kistler, Zach Moore, and Hamed Firooz. 2026. Large Scale Retrieval for the LinkedIn Feed Using Causal Language Models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 40. AAAI Press, Washington, DC, USA, 40101–40109.
- [10] Kaushik Rangadurai, Siyang Yuan, Minhui Huang, Yiqun Liu, Golnaz Ghasemiesfeh Ahsan, Yunzhong Tian, Siyu Lu, Xilun Zheng, Ming Hu, Haixin Liang, Qingwen Li, Fedor Cao, Xiaoqiang Liu, and Rong-En Sun. 2024. Hierarchical Structured Neural Network for Retrieval. arXiv:2408.06653
- [11] Sudarshan Srinivasa Ramanujam, Akanksha Bindal, Yu Jiang, Timothy J Hazen, David Golland, Fengyu Zhang, Daqi Sun, Wanning Li, Birjodh Singh Tiwana, Siddharth Dangi, et al. 2025. LinkedIn Post Embeddings: Industrial Scale Embedding Generation and Usage across LinkedIn. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*. 6038–6044.
- [12] Liangcai Su, Fan Yan, Jieming Zhu, Xi Xiao, Haoyi Duan, Zhou Zhao, Zhenhua Dong, and Ruiming Tang. 2023. Beyond Two-Tower Matching: Learning Sparse Retrievable Cross-Interactions for Recommendation. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '23)*. ACM, New York, NY, USA, 548–557. doi:10.1145/3539618.3591643
- [13] Bi Xue, Hong Wu, Lei Chen, Chao Yang, Yiming Ma, Fei Ding, Zhen Wang, et al. 2025. SilverTorch: A Unified Model-based System to Democratize Large-Scale Recommendation on GPUs. arXiv:2511.14881
- [14] Haoqiang Yang, Congde Yuan, Kun Bai, Mengzhuo Guo, Wei Yang, and Chao Zhou. 2025. HIT Model: A Hierarchical Interaction-Enhanced Two-Tower Model for Pre-Ranking Systems. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management (CIKM '25)*. ACM, New York, NY, USA. doi:10.1145/3746252.3761501
- [15] Jiaqi Zhai, Zhaojie Gong, Yueming Wang, Xiao Sun, Zheng Yan, Fu Li, and Xing Liu. 2023. Revisiting Neural Retrieval on Accelerators. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23)*. ACM, New York, NY, USA, 5520–5531. doi:10.1145/3580305.3599897
- [16] Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed Chi. 2019. Recommending what video to watch next: a multitask ranking system. In *Proceedings of the 13th ACM conference on recommender systems*. 43–51.