MixLM: High-Throughput and Effective LLM Ranking via Text-Embedding Mix-Interaction

Guoyao Li Ran He* Shusen Jing* Kayhan Behdin* Yubo Wang*
Sundara Raman Ramachandran* Chanh Nguyen* Jian Sheng* Xiaojing Ma* Chuanrui Zhu*
Sriram Vasudevan Muchen Wu Sayan Ghosh Lin Su Qingquan Song† Xiaoqing Wang
Zhipeng Wang Qing Lan Yanning Chen Jingwei Wu Luke Simon† Wenjing Zhang
Qi Guo‡ Fedor Borisyuk‡
{guoyli, rahe, sjing, kbehdin, yubwang, suramachandran, cnguyen, jsheng, xjma, chuzhu, svasudevan, muwu, sayaghosh, lsu, qsong, xiaoqwang, zhipwang, qlan, yannchen, jingwu, lsimon, wzhang, qguo, fborisyuk}@linkedin.com
LinkedIn, Mountain View, CA, USA

Abstract

Large language models (LLMs) excel at capturing semantic nuances and therefore show impressive relevance ranking performance in modern recommendation and search systems. However, they suffer from high computational overhead under industrial latency and throughput requirements. In particular, cross-encoder ranking systems often create long context prefill-heavy workloads, as the model has to be presented with the user, query and item information. To this end, we propose **MixLM**, a novel LLM-based ranking framework, which significantly improves the system throughput via reducing the input context length, while preserving the semantic strength of cross-encoder rankers. In contrast to a standard ranking system where the context is presented to the model as pure text, we propose to use **mix-interaction**, a mixture of text and embedding tokens to represent the input. Specifically, MixLM encodes all items in the catalog into a few embedding tokens and stores in a nearline cache. The encoded item descriptions are used during online inference, effectively reducing the item length from a few thousand text tokens to a few embedding tokens. We share insights from deploying our MixLM framework to a real-world search application at LinkedIn, including a detailed discussion of our training pipelines, as well as a thorough analysis of our online serving infrastructure optimization. Comparing with strong baselines, MixLM increased throughput by 10.0x under the same latency budget, while maintaining relevance metrics. The efficiency gains delivered by MixLM enabled full-traffic deployment of LLM-powered search, which resulted in a significant 0.47% increase in Daily Active Users (DAU) in online A/B tests.

ACM Reference Format:
Guoyao Li Ran He* Shusen Jing* Kayhan Behdin* Yubo Wang*, Sundara Raman Ramachandran* Chanh Nguyen* Jian Sheng* Xiaojing Ma* Chuanrui Zhu* Sriram Vasudevan Muchen Wu Sayan Ghosh Lin Su Qingquan Song† Xiaoqing Wang Zhipeng Wang Qing Lan Yanning Chen Jingwei Wu Luke Simon† Wenjing Zhang, Qi Guo‡ Fedor Borisyuk‡, {guoyli, rahe, sjing, kbehdin, yubwang, suramachandran, cnguyen, jsheng, xjma, chuzhu, svasudevan, muwu, sayaghosh, lsu, qsong, xiaoqwang, zhipwang, qlan, yannchen, jingwu, lsimon, wzhang, qguo, fborisyuk}@linkedin.com, LinkedIn, Mountain View, CA, USA . 2025. MixLM: High-Throughput and Effective LLM Ranking via Text-Embedding Mix-Interaction. In *Proceedings of Preprint*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Preprint, Arxiv

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

1 Introduction

Large Language Models (LLMs) have been widely adopted across industrial applications due to their strong capabilities in both generative and predictive tasks. In particular, search and recommender systems have emerged as major application areas where LLMs deliver substantial quality improvements over existing approaches [11, 23, 25, 34]. For example, when used as a relevance judge, an LLM can interpret subtle linguistic cues and capture conceptual relationships, enabling it to retrieve and rank semantically relevant results that traditional lexical methods may overlook.

Despite their impressive text-understanding capabilities, deploying large language models (LLMs) in search and recommender systems remains challenging due to their substantial computational overhead, particularly when processing long text inputs, making it difficult to meet the stringent industrial requirements for high throughput and low latency. As an example, a standard (cross-encoder) approach to using LLMs for ranking and search applications is to form an input text prompt, including user's query and candidate item(s) description, that instructs the LLM to determine if the query and the item are relevant [22–25]. This requires the LLM to process long input prompts. Such workloads with heavy *prefill* often lead to large latency, making them unacceptable for real-world industrial systems, as a result of the quadratic complexity of the self-attention mechanism, a key component of LLM architectures, with regards to the input context length. Often, in real-world implementation of LLM-based semantic search and recommender systems, the LLM has to be replaced with a smaller model, or the

*Equal key contributors.
†Work done while at LinkedIn.
‡Correspondence authors.

features that lead to long input prompts have to be dropped from the online serving stack [6, 29, 32]. Thus, reducing the input context length for LLM-based ranking and search applications remains an important research problem.

In this work, we introduce **MixLM**, an LLM-based ranking framework designed to substantially reduce input context length for semantic search applications while preserving the semantic richness of full-text. MixLM operates by compressing each candidate document, often containing thousands of tokens, into a compact set of learned embedding tokens using an LLM encoder. These embeddings are stored in a nearline cache and, during online inference, are mixed with the user's natural language query (and auxiliary textual signals) and processed by a ranker LLM. By reducing the document representation from thousands of tokens to a handful of embedding tokens, this mixed-input design achieves *orders-of-magnitude lower computational complexity*, while retaining the rich query–document interactions characteristic of full-text LLM rankers.

Beyond the core modeling approach, we present insights from developing and deploying MixLM in LinkedIn's production semantic job search system. We describe how to jointly train the encoder and ranker models in an end-to-end manner, and we introduce new distillation losses that effectively transfer knowledge from an existing full-text LLM ranker into the mixed-interaction architecture, ensuring alignment between embedding-based and text-based ranking behaviors. We further study scaling properties of MixLM with respect to training data size and the number of embedding tokens per item.

From a systems perspective, we provide a comprehensive analysis of MixLM's impact on serving infrastructure. By reducing the context length through embedding compression, MixLM yields substantially higher ranking throughput. We detail optimization techniques tailored for mixed-input LLMs, including nearline caching and amortized prefill computation, and show how these techniques compound the efficiency gains.

Finally, we demonstrate MixLM's effectiveness at industrial scale through its deployment in LinkedIn's AI-driven Job Search. MixLM:

- improves serving throughput by **10.0**× under the same latency budget,
- preserves the relevance quality of a strong full-text LLM baseline.

Combined with its efficiency advantages, MixLM enabled full LLM-ranking-based search deployment and delivered a significant **0.47%** lift in Daily Active Users (DAU) in a large-scale online A/B experiment.

Summary of Contributions:

- We present **MixLM**, an industrial-scale ranking framework that aims to reduce the prompt length for ranker LLMs by substituting candidate items' textual description with a few tokens of embeddings.
- We introduce an end-to-end training pipeline that jointly optimizes the encoder LLM and the ranker LLM, that allow for capturing rich query–item–feature interactions beyond the capacity of bi-encoder architectures.
- We describe a production-ready serving stack that incorporates several system optimizations pertaining to our mixed input ranking system, achieving an order-of-magnitude improvement in latency and GPU throughput.
- We share lessons and insights from developing and deploying our MixLM framework online to LinkedIn's semantic job search, the primary job search experience on the platform.

Taken together, these advances make LLM-based ranking practically deployable and economically sustainable on the industrial scale, enabling search and recommender systems that demand both deep language understanding and real-time responsiveness.

2 Semantic Search

Semantic Search. refers to retrieving and ranking items based on their *semantic similarity* to a given query rather than exact lexical overlap. A typical pipeline includes (1) *representation* of queries and items into dense embeddings using neural encoders, (2) *retrieval* of top-k candidates via exhaustive vector search—for example, GPU-accelerated brute-force search, (3) *reranking* of the retrieved candidates using a higher-capacity model that jointly considers query–item interactions, and (4) *application of business logic* such as personalization and diversity.

In this work, we focus on the core ranking problem within this semantic search framework. Specifically, we adopt a pointwise ranking structure that takes as input the textual information of the query and item (and additional user profile information, if appropriate), and predicts either the relevance between the query and item or the likelihood of user engagement. A product policy specifies how to grade the relevance of each (query, job) pair. The ranking system is then trained to optimize Normalized Discounted Cumulative Gain at the tenth position (NDCG@10) using policy-provided labels.

More concretely, given a query q and a candidate item j, we aim to predict how relevant the pair (q, j) are. In a basic cross-encoder ranking framework, one can be create a textual prompt, taking the form:

$$\text{prompt}(q, j) = \text{system prefix}, q, j \tag{1}$$

where the system prefix include chat template tags and instructions to the LLM to determine if the job posting j is a good match to the query q. Given that the relevance problem studied here is a binary classification problem*, this textual prompt can be passed through an LLM with an appropriate classification output head, to estimate the relevance score. Particularly, the LLM (with a binary classification head) can estimate the probability distribution $(p_{\text{yes}}, p_{\text{no}})$ where a higher value of p_{yes} indicates a higher likelihood of relevance match:

$$p_{\text{yes}}(q, j) = \mathbb{P}(j \text{ is relevant to } q) \in [0, 1]. \tag{2}$$

LinkedIn Semantic Job Search. In this work, we focus on LinkedIn's Semantic Job Search, which redefines job discovery through an AI-driven focus on intent rather than keywords. By interpreting natural language inputs, it captures users' underlying goals and preferences, minimizing dependence on exact word matches between queries and job descriptions. This semantic understanding allows for more accurate and flexible retrieval, overcoming vocabulary gaps and delivering search results that align more closely with how members naturally express their career interests.

*In this work, we do not study setwise or listwise ranking frameworks.

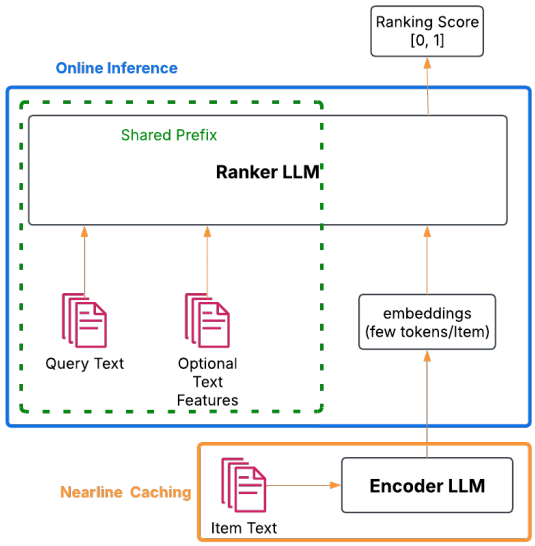

Figure 1: Architecture of MixLM. In the offline stage, item documents are processed by an encoder LLM to obtain compact embedding representations, which are stored in a near-line cache. At serving time, these embeddings are fetched and concatenated with the user's query and optional auxiliary text features, and the resulting mixed input is passed to the ranker LLM to generate relevance scores.

In practice, the prompt in (1) can be lengthy. In fact, in our semantic job search system, most of the prompt will be dominated with item description j, which has a median length of 900 tokens, and can take up to 2100 tokens. The lengthy prompts lead to unacceptable latency and throughput overhead, which makes deploying such a text-based system to production costly. To handle job search traffic spanning various product surfaces, our ranking system must process and score about 3.15 million items per second, making ranking efficiency absolutely critical.

Even though through previous summarization-solution[3] we were able to reduce the item description length and rollout LLM ranking in limited-scale traffic, the prompt context length remains the major computational bottleneck and hard blocker for LinkedIn's Semantic Job Search's full deployment.

3 Architecture of MixLM

As the computational complexity of LLM inference scales quadratically with input length, reducing the number of tokens can substantially lower inference costs. In this work, we propose a text–embedding mix-interaction architecture that compresses item text description into an embedding vector using an encoder LLM. The resulting embeddings are then combined with other textual inputs and fed into a ranker LLM for output prediction. Because the item embeddings are precomputed and stored in a near-line cache, the number of tokens that must be processed during online inference is greatly reduced, leading to significant efficiency gains.

Detailed Structure. Our ranking architecture consists of two components: an encoder LLM, which computes embeddings that summarize item information, and a ranker LLM, which predicts the final task scores. See Figure 1 for an overview.

Before diving deeper into our model architecture, let us recall that an LLM can be decomposed into three main components: The input embeddings layer, the transformer blocks and the optional output head. For an LLM, the input embeddings layer creates the input's embedding representation. The transformer layers then form the output embedding representations, given the input embeddings. Finally, the LLM output is generated via an optional output head. For example, for a relevance classification model (in contrast to generative LLMs), we might assume the output head is a binary classification head which generates the estimated probability that the query and item are relevant.

For a sequence of length T and a fixed set of tokenized vocabulary $\mathbb{V}$, we let $g_R(\cdot;\omega_R) : \mathbb{V}^T \to \mathbb{R}^{T\times H}$ to denote the input embedding layer of the Ranker LLM, parameterized by ω_R, with a hidden dimension of H. We also let $f_R(\cdot;\theta_R) : \mathbb{R}^{T\times H} \to [0,1]$ denote the Ranker LLM decoder and output head, parameterized by θ_R, where the output quantifies the probability of query-item match (i.e., p_{yes}). The Ranker LLM can thus be expressed as

$$F_R(\cdot;\Theta_R) = (f_R \circ g_R)(\cdot;\omega_R,\theta_R) : \mathbb{V}^T \to [0,1], \tag{3}$$

where $\Theta_R = [\omega_R, \theta_R]$ represents all ranker parameters.

Similarly, let $\Theta_E = [\omega_E, \theta_E]$ denote the parameters of the Encoder LLM. The Encoder LLM for a sequence of length T can be parameterized as

$$F_E(\cdot;\Theta_E) = (f_E \circ g_E)(\cdot;\omega_E,\theta_E) : \mathbb{V}^T \to \mathbb{R}^{T\times H}, \tag{4}$$

where $g_E(\cdot;\omega_E) : \mathbb{V}^T \to \mathbb{R}^{T\times H}$ is the embedding layer and $f_E(\cdot;\theta_E) : \mathbb{R}^{T\times H} \to \mathbb{R}^{T\times H}$ is the transformer that maps token embeddings to hidden representations (note that we do not have an output head for the encoder model).

In our mix-interaction system, we decompose the prompt in (1) into two parts, corresponding to ranker and encoder parts, and tokenize each part:

$$\begin{aligned} X_R &= \text{tokenize}([\text{system prefix}, q]) \in \mathbb{V}^{T_R} \\ X_E &= \text{tokenize}([\text{prefix}, j]) \in \mathbb{V}^{T_E} \end{aligned} \tag{5}$$

where T_R (T_E) is the length of X_R (X_E). Note that X_E does not depend on the query q but only the item description j, and therefore, we can encode and store X_E in an offline cache. We then let $h_E = F_E(X_E) \in \mathbb{R}^{T_E\times H}$ to be the encoded item. To compress this item representation, we sample h_E to a shorter sequence of length T_S (specifically in MixLM, we use last N sampling), yielding $h_S = \text{Samp}(F_E(X_e)) \in \mathbb{R}^{T_S\times H}$. In general, one can consider several sampling techniques. In our work, for a give $T_S \geq 1$, we sample the T_S embeddings in h_E corresponding to the last T_S input tokens. In particular, if $T_S = 1$, we only take the embedding representation of the last input token.

Next, the input token embeddings of the prefix X_R are computed via the ranker embedding layer, i.e., $h_R = g_R(X_R)$. The two representations h_R, h_S are concatenated to form the combined embedding sequence $[h_P; h_S] \in \mathbb{R}^{(T_R+T_S)\times H}$. The final "yes" probability is predicted by the ranker transformer:

$$p_{\text{yes}}(q, j; \Theta) = f_R([h_P; h_S]) \in [0,1], \tag{6}$$

where $\Theta = [\Theta_R, \Theta_E]$ denotes the full set of trainable parameters.

Since $h_S = \text{Samp}(F_E(X_e))$ can be precomputed and stored in a near-line cache, the effective sequence length during online inference is reduced from $T_R + T_E$ to $T_R + T_S$. In practice, $T_S \ll T_E$, where we might take $T_S = 1$ and T_E can be up to 2100, representing orders of magnitudes of reduction in the prompt length. Furthermore, when ranking multiple items, X_R is shared across all prompts that correspond to the same user and query. As the result, the Key–Value (KV) cache of X_R of be used leading to further reduction of computational complexity.

4 Training

In this section, we describe the models, datasets and training recipes for the **MixLM** framework deployed to LinkedIn's semantic job search system.

4.1 Training Overview

Our MixLM framework requires us to train both the ranker and encoder LLMs. In particular, we require the encoded item embeddings available from the encoder model to align with the ranker model's input embeddings, so the ranker model can make use of the encoded information. In general, as long as the ranker LLM's hidden size is the same as encoder's final output size (which allows us to concatenate embeddings in (6)), they can differ in architecture. However, to streamline the training process, we use models that have the same architecture for ranker and encoder, each with 0.6B parameters. We follow a three-stage training recipe for MixLM, summarized in Table 1. In stage I, we perform supervised fine-tuning on a 0.6B pretrained model using a domain-specific dataset (details to follow). The model resulting from this stage acts as the base model for our ranker model training. In stage II, using a purely text prompt, we train a ranker model using the labeled ground truth, which serves as a *teacher* model for our third training stage. This is motivated by our experience that presenting the model with complete textual item descriptions results in high ranking quality. In stage III, we co-train the ranker and the encoder models using custom loss functions that allow us to distill the text-based model from the second stage into a mixed input model. We use the model from the first training stage as the initial model checkpoint for the second stage and the ranker in the third stage, and a 0.6B General Text Embedding (GTE) model trained using contrastive training as the initial encoder checkpoint in the third stage.

4.2 Datasets

4.2.1 Label Generation. We sampled query–item pairs from real user logs, and train a 7B LLM as our relevance judge to generate 5-point graded scores along several dimensions, together with rationales, such that it aligns with the product policy. The integer labels were subsequently normalized to continuous values in the range $[0, 1]$ to create the ground truth labels $p^*_{\text{yes}}(q, j)$ for query-job pairs.

This 7B in-house LLM operates using the following template:

```
[CRITERIA]   : <matching guidelines>
[EXAMPLES]   : <reasoning and output format>
[QUERY]      : <query text>
[CANDIDATE]  : <item text>
```

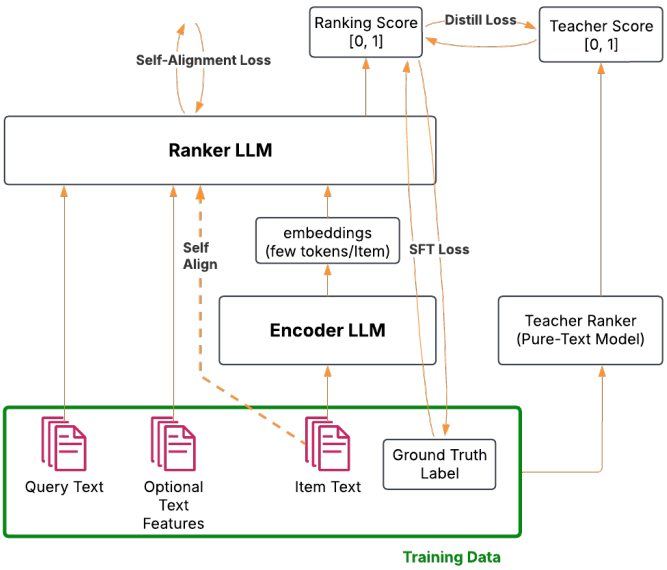

Figure 2: Detailed training pipeline of training state III.

```
Analyze the query--candidate pair and determine whether
they are a good match. Return a single matching score
(0/1/2/3/4) with detailed reasoning.
```

We gather model's chain-of-thought traces [33], as well as the final matching score which we use in different stages of training.

4.2.2 Teacher Training Prompts in Stage II. For the textual teacher training, we applied the following chat template to the ranker LLM:

```
[SYSTEM]    : Output only 'Yes' or 'No' based on how well
              the job matches the query; no extra text.
[QUERY]     : <query text>
[CANDIDATE] : <item text>
yes or no ?
```

4.2.3 Mixed Input Ranking Training Prompts in Stage III. For the joint ranking training (co-train the encoder LLM and ranker LLM), we applied the following chat template to the ranker LLM:

```
[SYSTEM]    : Output only 'Yes' or 'No' based on how well
              the job matches the query; no extra text.
[QUERY]     : <query text>
[CANDIDATE] : <item embeddings>
yes or no ?
```

where the `<item embeddings>` is the output from encoder LLM. The encoder LLM use the following prompt "`Item information: <item text>`". Note that the ranking prompt used here is significantly shorter compared to the teacher one in Section 4.2.2 which uses full item text.

4.3 Three-stage Training

Next, we provide the details of our training pipeline.

4.3.1 Stage I: Domain Reasoning Fine-Tuning. In the first stage, we train a general pretrained LLM using a domain-specific reasoning

Table 1: MixLM Training Stages: In the stage I, we use the chain-of-thought reasoning traces from our in-house relevance judge to pretrain a 0.6B model on the domain-specific data. In stage II, we apply Supervised Fine-Tuning (SFT) on the model from stage I with the text-only ranking dataset, which serves as a teacher model in the next stage. In stage III, we co-train the encoder and ranker LLMs using a ranking dataset. We include several different loss functions in this stage, including the ranking SFT loss, a distillation loss and a self-alignment regularization.

Stages	Stage I: Domain-specific Fine-Tuning	Stage II: Ranking Teacher Training	Stage III: Joint Encoder-Ranking Training
Data	Reasoning Dataset	Ranking Dataset	Ranking Dataset
Number of Samples	180k	10.9M	10.9M
Input Prompt Type	Text (chain-of-thoughts)	Text	Text + Embedding
Maximum Sequence Length	2 048	2 048	2 176
Trainable Module(s)	Pretrained LLM	Ranker LLM	Encoder LLM & Ranker LLM
Training Objective	Distillation from relevance judge	SFT	SFT, Distillation from ranking teacher, self-alignment

dataset to strengthen its capability for logical inference and contextual understanding over query–item pairs. This intermediate training phase enables the model to grasp the rationale behind assigning relevance scores, thereby enhancing its generalization performance in downstream ranking tasks.

Concretely, we distill [9] our 7B relevance judge into a 0.6B pretrained model, using KL divergence between the logits of the two models. As for the data, we randomly select 180K samples following the prompt given in Section 4.2.1, together with the chain-of-thoughts responses of the relevance judge model.

4.3.2 Stage II: Ranking Teacher Training. In the next step, we train a purely textual ranking model. Although this intermediate model cannot be served online due to the long input prompts, it will serve as a teacher model when training the mixed input MixLM model. As we demonstrate, using such a ranking teacher model results in better convergence and training quality.

Specifically, starting from the checkpoint available from our first training stage, we perform Supervised Fine-Tuning (SFT) using the input prompt discussed in Section 4.2.2. We generate 10.9M examples based on the prompt provided above, and normalize the integer matching scores, available from the 7B judge, into ground truth probabilities $(p^*_{\text{yes}}, p^*_{\text{no}})$. The SFT is then performed using a KL divergence loss as in (**??**). We use $(\hat{p}_{\text{yes}}, \hat{p}_{\text{no}})$ to denote the output probability distribution from the final checkpoint for this training stage.

4.3.3 Stage III: Joint Encoder-Ranking Training. In the final training stage, we train both ranker and encoder models to estimate relevance matching scores, using the mixed input prompt from Section 4.2.3. Specifically, the relevance scores are calculated using (6). The trainable parameters are $\Theta = [\Theta_R, \Theta_E]$ which encompasses all parameters of ranker and encoder models.

To this end, we initialize the ranker module in Figure 1 via the final checkpoint from our first stage training. For the encoder module, we use a pretrained 0.6B GTE model. We use the same 10.9M sample dataset from the previous stage for training. Given the specific structure of the ranking stack, we have observed using custom loss functions can improve the training quality and the final models' ranking quality. Below we discuss the details of the loss functions we use.

SFT Loss. Our first loss function is a KL divergence SFT loss, that ensures the predicted probabilities from MixLM are close to the ground truth. This loss function is given as

$$\mathcal{L}_{\text{SFT}}(\Theta) = \text{KL}\left((p^*_{\text{yes}}, p^*_{\text{no}})||(p_{\text{yes}}(\Theta), p_{\text{no}}(\Theta))\right)$$

where $(p_{\text{yes}}, p_{\text{no}})$ are the predicted probability of relevance match (see (6)), and $(p^*_{\text{yes}}, p^*_{\text{no}})$ are ground truth probabilities.

Ranking Distillation Loss. In addition to comparing to the ground truth, we compare the output distribution from MixLM to the one from the ranking teacher model from Stage II. More specifically, we define

$$\mathcal{L}_{\text{distill}}(\Theta) = \text{KL}\left((\hat{p}_{\text{yes}}, \hat{p}_{\text{no}})||(p_{\text{yes}}(\Theta), p_{\text{no}}(\Theta))\right)$$

where we recall that, $(\hat{p}_{\text{yes}}, \hat{p}_{\text{no}})$ is the output distribution of ranking teacher with full text prompt in Section 4.2.2.

This loss function is inspired by knowledge distillation [9], where knowledge from a (typically larger) teacher model is transferred to a (usually smaller) student model. In contrast, our student model, MixLM, which consists of an encoder and a ranker, can in fact be larger than the teacher model. As discussed in Section 6, the teacher operating on pure text is easier to train to provide high-quality predictions that are cleaner than the original dataset labels. When training MixLM, the teacher label helps reducing gradient variance and facilitates better convergence.

Self-Alignment Loss. Finally, we use two self-alignment loss functions, that help to ensure the encoder model's output embeddings are aligned with the input embeddings of the ranker model. From our experience, including these loss functions has a regularizaiton effect that can prevent overfitting. To this end, we pass the full text prompt (from Section 4.2.2) through the mixed input model's ranker module (the model parameterized by Θ_R from section 4), and collect the output probability distribution (which we denote by $(\bar{p}_{\text{yes}}, \bar{p}_{\text{no}})$). Additionally, we collect the last layer's (before the output head) hidden states corresponding to the last input token, which we denote by $\bar{h}_{\text{last}} \in \mathbb{R}^H$. Note that these quantities are functions of only Θ_R. For the same sample, we also collect the output probability distribution (denoted as $(p_{\text{yes}}, p_{\text{no}})$) and the last layer's hidden states for the last input token (denoted as $h_{\text{last}} \in \mathbb{R}^H$) when we use the mixed input prompt from Section 4.2.3. These quantities, as expected, are functions of all trainable parameters Θ. Then, we use loss functions

that encourage the resulting quantities from these two setups to be similar. First, we use a loss function $\mathcal{L}_{\text{hidden-align}}$ to encourage the last hidden states from the mixed input and textual prompts to be similar:

$$\mathcal{L}_{\text{hidden-align}}(\Theta) = 1 - \text{cossim}\big(h_{\text{last}}(\Theta), \bar{h}_{\text{last}}(\Theta_R)\big)$$

where cossim denotes the cosine similarity of two vectors. In addition, we use another alignment loss to ensure the predicted probabilities remain similar:

$$\mathcal{L}_{\text{pred-align}}(\Theta) = \text{KL}\left((\bar{p}_{\text{yes}}(\Theta_R), \bar{p}_{\text{no}}(\Theta_R))||(p_{\text{yes}}(\Theta), p_{\text{no}}(\Theta))\right).$$

The overall alignment loss is a linear combination of hidden state and prediction alignment losses:

$$\mathcal{L}_{\text{align}}(\Theta) = \alpha \cdot \mathcal{L}_{\text{pred-align}}(\Theta) \ + \ \beta \cdot \mathcal{L}_{\text{hidden-align}}(\Theta)$$

where $\alpha, \beta \geq 0$ are given hyper-parameters.

Finally, the full training objective integrates all components discussed above. Specifically, we use

$$\mathcal{L}_{\text{total}}(\Theta) = \mathcal{L}_{\text{SFT}}(\Theta) + \lambda_{\text{distill}}\mathcal{L}_{\text{distill}}(\Theta) + \lambda_{\text{align}}\mathcal{L}_{\text{align}}(\Theta),$$

where $\lambda_{\text{distill}} \geq 0$ and $\lambda_{\text{align}} \geq 0$ are hyper-parameters that are given a priori. Figure 2 summarizes the training pipeline for this stage.

Like multi-modal language models, such as LLaVA [21] and BLIP-2 [17], the goal of Stage III is to align the input space of the ranker LLM and the embedding space of the encoder LLM. This is challenging because 1) unlike the discrete vocabulary of text, item embeddings have an infinitely large continuous space, resulting in higher sample complexity; 2) the embeddings encode much more complex and entangled information than text tokens, requiring greater training effort to preserve and decode such information; and 3) item embeddings have never been encountered during foundational pretraining.

To train MixLM more effectively, we apply a phased curriculum learning strategy, decomposing the joint optimization into two distinct phases with different loss weights, i.e., (λ_{distill} and λ_{align}): In **Phase 1**, we prioritizes the space-alignment by increasing λ_{align} and decreasing λ_{distill}, so that we can train the encoder to produce embeddings that integrate seamlessly with the ranker's internal representations. In **Phase 2**, decrease λ_{align} and increase λ_{distill}, focusing on adjusting ranker LLM to improve the relevance score prediction.

This approach mirrors the multi-stage training strategies successfully employed in multi-modal LMs, where phased training with different optimization objectives in each phase improves both alignment quality and downstream task performance. During the training, we employ a standard AdamW optimizer, combined with a cosine annealing scheduler with linear warmup.

4.4 Training Efficiency

During the ranking training phase, we co-train the encoder LLM and the ranker LLM end-to-end. We deploy PyTorch's Fully Sharded Data Parallel [41] to shard the decoder parameters, gradients, and optimizer states across 8 nodes (each with 8 NVIDIA H100 GPUs). Within each node, GPUs communicate via NVLink, and inter-node communication is via InfiniBand.

To reduce communication overhead, we enable layer-wise prefetching: before computing the next forward layer, we prefetch its parameters; during the backward pass, we prefetch gradients of upcoming layers. To further reduce memory footprint, we incorporate Liger Kernel [12], a set of optimized Triton kernels (e.g. fused RMSNorm, RoPE, SwiGLU, fused CrossEntropy) that deliver higher throughput and lower memory usage.

In our setup, the total compute cost to train on 70B tokens under the ranking objective is approximately 700 H100 GPU-hours.

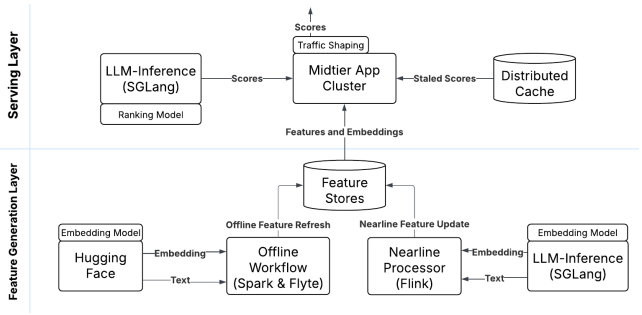

Figure 3: Online Serving Architecture for LinkedIn's Semantic Job Search

5 System Implementation

5.1 Online System

The online inference architecture adopts a two-layer design consisting of a feature generation layer and a serving layer (see Figure 3 for an overview). The feature generation layer functions as a hybrid system that combines offline refresh with near-line ingestion and generation to maintain comprehensive and up-to-date feature coverage. The offline pipeline, orchestrated through Flyte, periodically performs large-scale GPU-based inference to regenerate the full corpus of text and embedding features. In parallel, the near-line component continuously ingests real-time updates reflecting changes in the population of scoring candidates and generates corresponding features through GPU inference. Because near-line processing is not latency-sensitive, GPU resources can be utilized with high throughput and efficiency. All generated features are persistent in a high-throughput distributed storage system, ensuring consistency, low-latency availability for downstream serving.

The serving layer performs real-time inference by retrieving precomputed features and assembling mixed text–embedding prompts that integrate user queries with candidate scoring items. Prior to dispatching requests to the SGLang inference engine [42] for GPU-based scoring, several system-level optimizations—including distributed caching, latency reduction mechanisms, and scalable serving infrastructure—are applied to minimize feature retrieval overhead and improve serving efficiency. This layered architecture enables effective reuse of precomputed representations while preserving responsiveness across both near-line and real-time applications. Empirical evaluations demonstrate that these design choices significantly enhance inference efficiency, GPU utilization, and system scalability, supporting production-scale deployment of large language model–based ranking systems.

5.2 Inference Engine

5.2.1 Text-Embedding Mixed Input. Common open-source LLM inference engines focus on pure-text inputs, restricting requests to tokenized sequences. To enable mix-interaction in MixLM, we extended the input interface to additionally accept precomputed embedding tensors. These embeddings are transmitted using a structured *feature payload* abstraction added to the gRPC request schema. Each feature payload encapsulates a binary buffer containing tensor values, along with metadata specifying the tensor's shape, element type, and optional serialization format (e.g., NumPy [8]-compatible). This design allows clients to include dense embeddings alongside text inputs without modifying the text-based interface or introducing format-specific logic in the serving stack.

On the server side, the preprocessing module decodes and validates these feature payloads into native tensor objects, applying metadata-driven reshaping and casting as necessary. The resulting tensors are then attached to the model input context for downstream inference. This approach enables direct reuse of the embeddings computed by upstream retrieval or feature pipelines, avoiding redundant re-computation and reducing latency. The extended schema remains fully backward-compatible with existing text-only requests while introducing minimal overhead in serialization and deserialization.

5.2.2 Shared-Prefix Prefill Optimization. In large-scale search and recommendation platforms, a single ranking request typically contains hundreds to thousands of candidate items, all associated with the same query and user-history features. After we compress each item's description into a small, fixed set of embedding tokens, the computational burden of inference shifts predominantly to the repeating query/user-history side, because these tokens now constitute the majority of the prompt. For example, in our production system a candidate item was previously represented by roughly 900 tokens as median; with MixLM this representation is reduced to only two tokens—one special token and one embedding token.

To improve our GPU utilization during online inference, we use a batched inference mechanism. That is, for each query, we batch several prompts, each corresponding to a different candidate item, and run inference on the batch. Given that all prompts in a batch share the same prefix, this allows for reusing the KV cache across different prompts. This is particularly important when combined with the mixed-input system: once the item representation is reduced to just a few tokens, the shared query-prefix computation dominates the overall inference cost.

Let T_q and T_i denote the number of shared query tokens and the number of item tokens in the *full-text* representation, respectively, and let N_i denote the ranking depth. Each transformer block consists of a multi-head self-attention module and a set of linear layers (query/key/value projections and a feed-forward network). For a sequence of length L, we write the attention and linear costs using proportionality (ignoring architecture-dependent constants such as depth, number of heads, and hidden/FFN dimensions) as

$$\text{Self-attention cost} \;\propto\; L^2, \qquad \text{Linear/FFN cost} \;\propto\; L.$$

Thus, attention is quadratic in the sequence length L, while the linear layers are linear in L but may have a larger constant factor.

Naïve prefill cost (full-text items). Without amortization, each candidate item forms an independent sequence of length $(T_q + T_i)$. Across N_i items, the total attention and linear prefill costs scale as

$$F_{\text{att}}^{\text{naive}} \;\propto\; N_i(T_q + T_i)^2, \qquad F_{\text{lin}}^{\text{naive}} \;\propto\; N_i(T_q + T_i).$$

When both T_q and T_i are large, $F_{\text{att}}^{\text{naive}}$ contains dominant contributions from $N_i T_q^2$, $N_i T_i^2$, and $N_i T_q T_i$, while $F_{\text{lin}}^{\text{naive}}$ grows linearly in $(T_q + T_i)$.

Amortized prefill (full-text items). With *amortized prefill*, only the first sequence in a batch performs the full prefill over the query prefix; the remaining $N_i - 1$ sequences reuse its KV cache and do not recompute query self-attention. The attention cost decomposes into

$$F_{\text{att}}^{\text{amort, full}} \;\propto\; \underbrace{T_q^2}_{\text{query prefix once}} \;+\; \underbrace{N_i\big(2T_iT_q + T_i^2\big)}_{\text{items attending to prefix and themselves}},$$

while the linear layers are applied once to the query prefix and once to each item token:

$$F_{\text{lin}}^{\text{amort, full}} \;\propto\; T_q + N_iT_i.$$

Amortized prefill with MixLM compression. In MixLM, we use T_i to denote the length of the original full item text, and introduce a compression factor $K > 1$ such that the item is represented by T_i/K tokens in the mixed-input prompt. In our online A/B test, we obtained a compression factor of $K \approx 450$, i.e., each item is compressed from hundreds of tokens down to two tokens in the prompt.

Under amortized prefill with MixLM, the query prefix of length T_q is still processed once, but each item now contributes only T_i/K tokens. The attention cost becomes

$$F_{\text{att}}^{\text{amort, MixLM}} \;\propto\; T_q^2 \;+\; N_i\Big(2(T_i/K)T_q + (T_i/K)^2\Big),$$

and the linear-layer cost is

$$F_{\text{lin}}^{\text{amort, MixLM}} \;\propto\; T_q + N_iT_i/K.$$

Compared to the full-text setting, both the attention and linear components of the item-dependent cost are reduced by approximately a factor of K on the item side. In particular, the dominant cross-term in attention changes from $N_iT_iT_q$ to $N_i(T_i/K)T_q$, and the item-side linear cost changes from N_iT_i to $N_i(T_i/K)$.

To realize this amortization, we explored two approaches:

- **In-batch prefix caching** We reuse the key–value (KV) states from the first prompt so that all items in the batch share the same query-prefix computation in a single forward pass. The attention computation merges two operations:
  (1) suffix tokens attend to all prefix tokens from the first prompt via dense paged attention;
  (2) suffix tokens attend to themselves via regular causal attention.
- **Multi-item scoring** We score multiple items against a single query within one forward pass by concatenating the query and all item texts into a single sequence [30], separated by a special delimiter token. FlashInfer's item-aware masking ensures that each item attends only to the query and not to any other item. During prefill, metadata such as prefix length, item boundaries, and maximum item length is passed

Table 2: Attention and linear-layer compute under different conditions, in terms of (T_q, T_i, N_i), where T_i denotes the full item-text length and K is the MixLM compression factor (item length becomes T_i/K in the prompt). The symbol $\propto$ indicates proportionality up to architecture-dependent constants (layers, heads, hidden size, etc.). The bottom blocks instantiate the expressions for $N_i = 250$ and $K = 450$.

Conditions	Total attention cost	Total linear-layer cost
Naive (full-text items)	$\propto N_i(T_q + T_i)^2$	$\propto N_i(T_q + T_i)$
Amortized prefill (full-text items)	$\propto T_q^2 + N_i(2T_iT_q + T_i^2)$	$\propto T_q + N_iT_i$
Amortized prefill + MixLM (item length T_i/K)	$\propto T_q^2 + N_i\big(2(T_i/K)T_q + (T_i/K)^2\big)$	$\propto T_q + N_iT_i/K$
Naive (full-text items) with $N_i = 250$	$\propto 250(T_q + T_i)^2$	$\propto 250(T_q + T_i)$
Amortized prefill (full-text items) with $N_i = 250$	$\propto T_q^2 + 250(2T_iT_q + T_i^2)$ $= T_q^2 + 500T_iT_q + 250T_i^2$	$\propto T_q + 250T_i$
Amortized prefill + MixLM with $N_i = 250$, $K = 450$	$\propto T_q^2 + 250\big(2(T_i/450)T_q + (T_i/450)^2\big)$ $= T_q^2 + \frac{10}{9}T_iT_q + \frac{1}{810}T_i^2$	$\propto T_q + 250T_i/450$ $= T_q + \frac{5}{9}T_i$

directly to the FlashInfer kernel to build the correct attention mask and prevent cross-item dependencies.

Because T_i is extremely small in our mixed-input approach, amortized prefill dramatically reduces redundant prefix computation and yields substantial inference-throughput improvements in production.

5.2.3 CPU Overhead Reduction.

Multi-Process gRPC. After optimizing both the model and the SGLang engine, we observed that request preprocessing in the Python gRPC service became the new performance bottleneck. Although the gRPC server is implemented with asynchronous Python, tasks such as request deserialization, feature decoding, and context construction remain constrained by Python's Global Interpreter Lock (GIL) and the single-threaded nature of the asyncio event loop. High request volume can exacerbate these bottlenecks due to contention on the event loop and per-request overhead, limiting throughput despite asynchronous I/O. To overcome this constraint, we implemented a multiprocessing server architecture that decouples the gRPC server from the SGLang engine. This design enables preprocessing and request handling to run across multiple CPU cores in parallel, while the SGLang engine executes independently. As a result, this architecture yields a 40% improvement in end-to-end throughput.

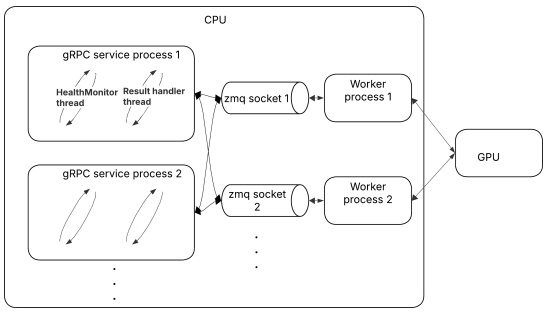

Figure 4: Multiprocessing server for gRPC

Batch Send. To better leverage amortized prefill, the SGLang scheduler must group prompts that share the same query. To address this, we introduced an explicit *batch send* pathway in which the entire set of item prompts for a query is serialized and sent as a single ZMQ message. This guarantees that the scheduler receives and processes the full batch together, allowing it to admit the batch as a unit and apply in-batch prefix caching efficiently.

Multi-Process Pallellization. Given our workload characteristics, the prefill phase completes rapidly on GPU, but request scheduling on a single CPU core became the bottleneck. We addressed this by deploying multiple serving engine's scheduler processes (e.g., 5 workers) sharing a single GPU instance, with each process allocated a dedicated memory partition (e.g., 20% GPU memory per worker). This parallelizes CPU scheduling overhead across multiple cores while maintaining full GPU utilization, effectively pipelining batch preparation with GPU execution.

5.2.4 Eliminating Redundant KV-Cache Operations.

SGLang's default behavior caches computed key-value pairs during prefill to enable efficient autoregressive decoding. However, our scoring use case is prefill-only with no decode phase. We modified the cache management to immediately free KV cache memory upon request completion rather than persisting it for non-existent decode operations. This eliminates unnecessary memory allocation, cache eviction logic, and pool management overhead on the critical path. More importantly, it increases effective GPU memory capacity, allowing higher batch sizes and concurrent request throughput.

6 Experimental Results

We perform offline and online experiments to understand the ranking quality and throughput improvements of our proposed framework. We use a held-out test set, that has been labeled using our internal LLM judge.

We compare our mixed input system with a few baselines. The first baseline is a bi-encoder embedding-based retrieval model, which has been distilled from our LLM relevance judge. This is the retrieval layer of our semantic search stack. The next baseline

Table 3: Rankin quality comparison of several ranking approaches: We report NDCG@10 with respect to the labels provided by our relevance judge. See Section 6 for a description of baselines.

Model	NDCG@10
Full Text	0.9432
Summarized, Pruned	0.9218
MixLM	0.9239
Embedding Retrieval	0.8380

Table 4: GPU efficiency comparison of ranking approaches. For each ranker, we report the observed maximum throughput (items scored per H100 GPU per second) under a fixed 500 ms latency budget. For reference, we also include the throughput of the retrieval layer (embedding-based exhaustive GPU retrieval, deployed on A100 GPUs). See Section 6 for details on the baseline methods.

Model	QPS(Items/Sec/GPU)	Latency (ms)	GPU
Full Text	290	<500	H100
Summarized, Pruned	2 200	<500	H100
MixLM	22 000	<500	H100
Embedding Retrieval	$> 1.6 \times 10^9$	<100	A100

is a pure-text system, that is, the 0.6B model available from our second training stage. This represents the ideal ranking model, though it is too inefficient to be deployed. In addition, we compare with LinkedIn's limited-scale production baseline [3] that uses a near-line text-summarization model to summarize item descriptions and apply aggressive pruning to the ranker model.

In terms of metrics, we report the NDCG@10 on the labeled held-out set. Additionally, we report the maximum throughput we can achieve for each baseline in our production setup. In particular, we report the maximum number of items we can score per GPU per second, while ensuring the P99 end-to-end latency remains below 500 milliseconds.

6.1 Comparisons with Baselines

Table 3 summarizes the ranking quality and Table 4 summarizes system efficiency of the baselines we discussed. Embedding-based retrieval, while able to process well over 1.6×10^9 documents per seconds on a single weaker A100 GPU, delivers the weakest ranking quality and is therefore usually employed only as a coarse first-stage filter. Feeding the full documents to a large language model (pure-text LLM) pushes NDCG to 0.9432, but its heavy token footprint limits throughput to merely 290 items/sec/GPU. Compressing each document into a short summary before prompting the LLM (summarized text LLM) and pruning ranker LLM's parameters by 37% [3] alleviates the latency bottleneck somewhat—throughput rises to 2,200 items/sec/GPU—yet ranking quality NDCG@10 is still an acceptable value as 0.9218. Our proposed MixLM achieves a similar NDCG@10 of 0.9239. However, we observe that MixLM allows us to scale up the throughput to 22,000 items/sec/GPU, i.e., a 10.0x speed-up over summarized-text LLM and a 75.9x speed-up over full-text LLM, at the cost of only a 1.8-point NDCG gap with respect to the latter. In other words, MixLM hits a sweet spot in the quality–efficiency trade-off: it is orders of magnitude faster than existing LLM-ranking baselines while retaining virtually the same retrieval effectiveness, making it an attractive drop-in replacement for latency-critical production systems.

Table 5: Online A/B Test

Experiment Group	DAU
Classic Job Search	–
Semantic Job Search (powered by MixLM)	**+0.47%**

6.2 Online A/B Test

Table 5 summarizes the findings. After rolling out MixLM in LinkedIn's AI Job Search and running extensive A/B tests, we observed: MixLM is has on-par metrics comparing with the existing limited-scale production baseline(summarized text LLM) [3]. With the 10.0x gain in serving throughput under the same latency budget, MixLM empowered the full-traffic rollout of LLM-ranking-based Semantic Job Search and for the first time, consequently producing a significant 0.47% increase in Daily Active Users, compared with the traditional job search.

6.3 Ablation Analysis

We conducted extensive ablation studies on a smaller subset that mirrors the distribution of the full training corpus, which is common practice in LLM training because full-scale experiments would have required high GPU hours. Within this setting, we systematically analyzed every major component of MixLM training—(1) dataset size, (2) number of token embeddings per item, (3) domain-reasoning fine-tuning, (4) auxiliary loss functions, and (5) curriculum learning strategies. Also, we conducted ablations on different inference engine optimization methods aimed at improving the computation of shared prefix prompt (query and the user-history).

6.3.1 Scaling Up. Expanding the amount of training data had a clear positive effect on ranking quality. In the ablation study, we progressively enlarged the corpus from 160 K to 1.08 M instances, observing a consistent increase in NDCG@10 at each step. Motivated by this trend, the model deployed in the online A/B experiment was trained on the full dataset of 10.9 M samples.

A similar monotonic gain was observed when we varied the representational granularity of each item. Raising the number of embedding tokens per item from 1 to 50 produced steady improvements in NDCG@10. Nonetheless, to meet strict latency requirements, the production configuration retains a single embedding token per item (followed by 1 special end-of-sentence text token).

6.3.2 Domain Reasoning Fine-Tuning. The ranker LLM serves as the backbone of our model architecture: both the query tokens and the compressed embedding tokens pass through this ranker to produce the final binary (yes/no) prediction. Consequently, the quality of the ranker LLM directly determines the overall ranking performance.

Table 6: Ablations on Dataset Size, on small sub-dataset

Num Training Sample	NDCG@10
160k	–
400k	+0.0250
800k	+0.0280
1.08M	+0.0334

Table 7: Embeddings Token Number per Item, on small sub-dataset

Tokens/item	NDCG@10
1	–
5	+0.0017
10	+0.0044
20	+0.0111
30	+0.0158
40	+0.0172
50	+0.0198

We found that employing a stronger base model for the ranker substantially improves the final results. Specifically, we compared two configurations: (1) using a open-source pretrained 0.6B model as the base, and (2) using a domain reasoning fine-tuned version of the same model. The results, summarized in Table 8, clearly show that the domain reasoning tuned base model yields superior bi-encoder performance.

Table 8: Effect of Domain Reasoning Fine-tuning on ranker LLM's Ranking performance, on small sub-dataset

Ranker Base Model	NDCG@10
vanilla open-source LLM	–
Domain-Reasoning Tuned LLM	**+0.0185**

Based on these findings, we have adopted Domain Reasoning Fine-Tuning as the first stage of our training recipe. In the first stage, we obtain a more powerful base model for the ranker LLM via task-specific COT data, guided by our internal 7B relevance judge LLM as the teacher model. After that, we use this tuned model as the Ranker LLM's starting base model, to initialize the training of stage III: Joint Ranking Training

6.3.3 Self-Alignment and Auxiliary Losses. To better understand the contribution of each auxiliary loss, we conducted an ablation study by selectively disabling different components. In all cases, we used the same setup (small sub-dataset, 50 embedding tokens per item) and varied the presence of distillation loss and self-alignment loss.

We observe that:

- Distillation loss contributes a significant improvement (+0.009 relative to KL-only).
- Self-alignment alone provides only marginal gains over the KL-only baseline.

Table 9: Ablation study on auxiliary losses: 50 embedding tokens per item, on small sub-dataset

Setup	NDCG@10
no auxiliary loss	–
+self-alignment loss	+0.0014
+distillation loss	+0.0091
+self-alignment +distillation losses	**+0.0108**

- Combining distillation and self-alignment achieves the best performance (NDCG@10 +0.0108), confirming that the two auxiliary losses are complementary.

These results highlight the importance of including distillation in addition to KL and self-alignment losses to maximize ranking quality.

6.3.4 Curriculum Learning Strategy. In addition to the auxiliary loss components, we investigated the effect of different phased training strategies that align with curriculum learning principles for Stage III in Section 4.3.3. Curriculum learning suggests that training with carefully sequenced objectives or loss weightings can lead to better convergence and final model performance. In the context of text-embedding alignment problem, we explored the following curriculum strategies:

- **No Curriculum Learning (Baseline)**: focus more on the KL divergence loss but less on distillation loss and self-alignment regularization.
- **Two-Phase Curriculum Strategy**: On top of the baseline, add an alignment phase at the beginning that has high self-alignment weights with low KL divergence weights.
- **Three-Phase Curriculum Strategy**: Add an intermediate phase to two-phase curriculum, that has equal weighting across all three loss objectives (distillation, SFT, and self-alignment).

As seen in Table 10, curriculum learning improves ranking performance, indicating the alignment is necessary. The two-phase curriculum outperforms the three-phase approach, suggesting that a direct transition between alignment-focused and task-performance-focused training is more effective than including an intermediate balanced phase. This finding indicates that once the ranker-encoder alignment is sufficiently established, the model can more directly optimize for the downstream ranking task without the gradual intermediate phase.

Table 10: Ablation study on curriculum learning strategies, on small sub-dataset

Curriculum Strategy	NDCG@10
no curriculum learning (Task)	–
two-phase (Alignment → Task)	**+0.0020**
three-phase (Alignment → Balanced → Task)	+0.0015

6.3.5 Inference Optimizations. To evaluate the effectiveness of our inference optimizations for the MixLM framework, we conducted an ablation study measuring maximum throughput (QPS, items per

Table 11: Ablation Study on Inference Optimizations (<500 ms latency)

Configuration	Prefix Optimization	QPS (Items/s/GPU)
Raw-text	None	270
	Multi-Job Scoring	275
	In-Batch Prefix Cache	290
Summarized, Pruned	None	1 650
	Multi-Job Scoring	2 100
	In-Batch Prefix Cache	2 200
MixLM	None	3 000
	Multi-Job Scoring	20 000
	In-Batch Prefix Cache	**22 000**

second per GPU), within the same p99 latency budget of <=500 ms. The experiments were performed on an NVIDIA H100 80GB HBM3 GPU with CUDA 12.8. The workload consists of a shared prefix of 60 text tokens (search query and user history) and a batch size of 50 candidate items. In different experiment settings, candidate items have different corresponding representations: full item description texts ("raw-text", 766 tokens/item, 38300 tokens/batch), summarized item description texts ("summarized-text", 145 tokens/job, 7250 tokens/batch), and ("Mix-LM", 1 embedding + 1 special token = 2 tokens/job, 100 tokens/batch). Each representation is evaluated without optimizations, with In-Batch Prefix Caching and with Multi-Job Scoring.

Table 11 presents the maximum items throughput per second per GPU, for p99 latency under 500ms. For each configuration, it demonstrates the impact of input compression and inference optimizations. MixLM with In-batch prefix caching achieves the highest throughput (22,000 QPS), contributing to the increase in 10.0x throughput. These results highlight the efficiency gains from compressing job descriptions and optimizing query processing, validated on an H100 GPU.

7 Related Works

7.1 LLMs for Recommender Systems

Recent years have seen a surge of interest in utilizing LLMs for search and recommender system applications. For example, several papers have reported that LLMs' understanding of natural language makes them powerful relevance judges. Moreover, LLMs are strong ranker which can extract users' preferences based on their past interactions [11]. This has prompted a long line of research, investigating how LLMs can be integrated to existing recommender system, or even used as standalone recommenders [34]. For example, [35] show that LLMs can be successful in solving the cold-start problem in recommender systems, where not much historical data is available for the user. The cold-start problem was further studied by [38] through the lens of instruction tuning. A similar approach was also used by [2]. [19] discuss using LLMs to predict user actions and preferences when the model is presented with the past data as a text prompt.

7.2 LLMs for Semantic Search

More related to our work, there is a long line of research on how LLMs can benefit semantic search applications. A common approach to enhance relevance ranking using LLMs is the bi-encoder approach [15, 22], where the query and the item are passed through two separate encoders (or *towers*) to obtain embedding representations, and the resulting representations are compared to judge the relevance. This approach allows for a computationally efficient implementation, and has been extensively utilized in practice [13]. However, this representation-based approach might not be able to capture nuanced semantic clues, due to limited interaction between queries and documents. In response, other work has studied late interaction representation-based models [16].

An alternative approach to using LLMs for semantic search is the cross-encoder approach, where a single model scores the relevance of a query and an item (or multiple items under a list-wise ranking scheme) in a single prompt. Cross-encoder systems with LLM backbones have proved to be strong scorers for ranking systems [23, 24, 28, 39, 43]. Unfortunately, serving LLM-based cross-encoders as online relevance scorers at an industrial scale can be prohibitively expensive in practice, as scoring numerous pairs of query-item using an LLM would require access to significant computational resources [7, 10, 32, 36].

To overcome the computational requirements of LLM-based cross-encoders, other papers have proposed alternative approaches to use LLMs in relevance scoring applications. For example, *LLM-as-judge* systems make use of LLMs as policy teachers by assigning graded relevance labels and rationales to query–item pairs. This allows for creating training datasets that can be used to distill the knowledge from LLMs to more efficient architectures [26, 27, 31, 32, 36, 37]. Another approach is *LLM-as-augmenter*, where LLMs generate paraphrased queries, summarize long documents, and synthesize hard negatives to enrich training data [3, 18, and references therein].

7.3 Industrial Application of LLMs for RecSys

In addition to understanding the methodological aspects of MixLM, we also focus on a real-world semantic search that is deployed at an industrial scale. Related to this, [32] discuss how an LLM can be used as a relevance judge. However, the LLM-based system is eventually distilled into a non-LLM architecture for online serving. Similar challenges have been reported by [6] and [29], where either the model or some of the text features need to be altered for online serving. This further motivates our work to study methods that improve the efficiency of LLM-based recommender systems. In a recent work, [37] study the distillation an LLM-based system into a BERT-type [5] model. Notably, for most these real-world applications discussed here, to meet efficiency goal, either a non-LLM model has to be deployed, or some features have to be removed. In contrast, in our real-world application, we deploy the LLM online, and all features are consumed by our encoder model to create item embedding representations.

7.4 Mixed Input LLMs for RecSys

Mixed-input LLMs for recommender systems has also been studied by some prior work. [40] propose to substitute parts of the input

text prompt with embeddings available from traditional (non-LLM) embedding systems. In another work, [4] propose to use embeddings to present user's past interactions, instead of a text prompt. We note that however, our work differs from the aforementioned papers in a few key areas. First, we study a different prediction problem compared to [4, 40] for a relevance scoring system, where nuanced semantic understanding is of high importance. In addition, we present concrete lessons for training our MixLM system, including custom loss and regularization functions that differentiate our work. For example, we show that how an existing pure-text system can be distilled into the mixed-input framework using the custom loss functions we introduce. Finally, [40]'s analysis is limited to offline benchmarks, while we deploy our mixed input framework online for an A/B test. Moreover, even though Chen et al. [4] discuss online A/B testing, their hierarchical framework is used offline to generate user and item features, which are then fed to an online recommender system, and therefore, the LLM is not deployed for online inference. This is different from our work where the LLM is used for online ranking. Hence, we present lessons from optimizing the serving infrastructure for such a mixed input system, as well as providing online quality metrics.

Concurrent to our work, [20] studies a mixed input system for Retrieval-Augmented Generation (RAG) applications, which is a different problem from ours. Additionally, we study a real-world deployment of our system, while the analysis of [20] is mostly contained to an offline study. In particular, we present several lessons and insights from serving infrastructure optimizations.

7.5 Multimodal Large Language Model Training

A common way to build multimodal large language models (MLLMs) is to couple a pretrained text-only LLM with modality-specific encoders via lightweight connectors. Early systems such as Flamingo [1] and Kosmos-1 [14] showed that treating visual features as additional "tokens" for a transformer backbone enables strong performance on captioning, VQA, and open-ended dialogue, typically by inserting cross-attention layers or training a unified multimodal transformer on large image–text corpora.

Subsequent work has emphasized reusing vision encoders and LLMs, sometime adding a small bridge between them. BLIP-2 [17] uses a Querying Transformer to align vision features with the LLM embedding space, and LLaVA-style models [21] learn a simple projection that maps CLIP features directly into the token embedding space of an LLM, followed by visual instruction tuning. In all these approaches, the key idea is to bypass explicit tokenization of non-text inputs and inject continuous modality-specific embeddings into the LLM, preserving its architecture and linguistic knowledge while adding multimodal capability. Our MixLM follows this paradigm, directly feeding learned embeddings into the language model to replace the tokenizer and embedding layer.

8 Conclusion

In this work, we introduced MixLM, a novel framework that bridges the gap between the semantic richness of large language model–based rankers and the stringent efficiency requirements of industrial search and recommender systems. By representing candidate items as compact embedding tokens and introducing the mix-interaction mechanism, MixLM effectively reduces the input context length while maintaining the expressiveness of full-text LLMs. Our co-training strategy for online rankers and near-line embedding flows further enables tight coupling between semantic understanding and serving efficiency.

Empirical results from deployment in a large-scale professional social network demonstrate that MixLM delivers substantial system gains—10.0x times improvement in throughput, under same latency budget. MixLM enabled LLM search system's deployment on all LinkedIn users' traffic, thereby producing a measurable 0.47% increase in daily active users (DAU). These findings highlight the potential of mixed-input architectures to reshape the design space of LLM-based ranking systems, offering a practical path toward scalable, low-latency, and semantically powerful search in real-world environments.

Future work will explore adapting co-trained embeddings for modeling member history and exploring MixLM for retrieval tasks.

Acknowledgments

To all who collaborated with us to create Mix-LM.

References

[1] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. 2022. Flamingo: a visual language model for few-shot learning. *Advances in neural information processing systems* 35 (2022), 23716–23736.

[2] Keqin Bao, Jizhi Zhang, Yang Zhang, Wenjie Wang, Fuli Feng, and Xiangnan He. 2023. Tallrec: An effective and efficient tuning framework to align large language model with recommendation. In *Proceedings of the 17th ACM conference on recommender systems*. 1007–1014.

[3] Kayhan Behdin, Qingquan Song, Sriram Vasudevan, Jian Sheng, Xiaojing Ma, Z Zhou, Chuanrui Zhu, Guoyao Li, Chanh Nguyen, Sayan Ghosh, et al. 2025. Scaling Up Efficient Small Language Models Serving and Deployment for Semantic Job Search. *arXiv preprint arXiv:2510.22101* (2025).

[4] Junyi Chen, Lu Chi, Bingyue Peng, and Zehuan Yuan. 2024. Hllm: Enhancing sequential recommendations via hierarchical large language models for item and user modeling. *arXiv preprint arXiv:2409.12740* (2024).

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 4171–4186.

[6] Soumik Dey, Benjamin Braun, Naveen Ravipati, Hansi Wu, and Binbin Li. 2025. LLMDistill4Ads: Using Cross-Encoders to Distill from LLM Signals for Advertiser Keyphrase Recommendations at eBay. *arXiv preprint arXiv:2508.03628* (2025).

[7] Luyu Gao, Zhuyun Dai, and Jamie Callan. 2020. Understanding BERT rankers under distillation. In *Proceedings of the 2020 ACM SIGIR on International Conference on Theory of Information Retrieval*. 149–152.

[8] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. doi:10.1038/s41586-020-2649-2

[9] G Hinton. 2014. Distilling the Knowledge in a Neural Network. In *Deep Learning and Representation Learning Workshop in Conjunction with NIPS*.

[10] Sebastian Hofstätter, Sophia Althammer, Michael Schröder, Mete Sertkan, and Allan Hanbury. 2020. Improving efficient neural ranking models with cross-architecture knowledge distillation. *arXiv preprint arXiv:2010.02666* (2020).

[11] Yupeng Hou, Junjie Zhang, Zihan Lin, Hongyu Lu, Ruobing Xie, Julian McAuley, and Wayne Xin Zhao. 2024. Large language models are zero-shot rankers for recommender systems. In *European Conference on Information Retrieval*. Springer, 364–381.

[12] Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. 2024. Liger kernel: Efficient triton kernels for llm training. *arXiv preprint arXiv:2410.10989* (2024).

[13] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. 2020. Embedding-based retrieval in facebook search. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2553–2561.

[14] Shaohan Huang, Li Dong, Wenhui Wang, Yaru Hao, Saksham Singhal, Shuming Ma, Tengchao Lv, Lei Cui, Owais Khan Mohammed, Barun Patra, et al. 2023. Language is not all you need: Aligning perception with language models. *Advances in Neural Information Processing Systems* 36 (2023), 72096–72109.

[15] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick SH Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering.. In *EMNLP (1)*. 6769–6781.

[16] Omar Khattab and Matei Zaharia. 2020. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 39–48.

[17] Junnan Li, Dandan Li, Shlok Vaidya Suhartono, Yejin Song, Hao Bao, Yejin Zhang, et al. 2023. BLIP-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. In *International Conference on Machine Learning*. PMLR, 19730–19742.

[18] Xiaopeng Li, Xiangyang Li, Hao Zhang, Zhaocheng Du, Pengyue Jia, Yichao Wang, Xiangyu Zhao, Huifeng Guo, and Ruiming Tang. 2024. SyNeg: LLM-Driven Synthetic Hard-Negatives for Dense Retrieval. *CoRR* (2024).

[19] Jianghao Lin, Rong Shan, Chenxu Zhu, Kounianhua Du, Bo Chen, Shigang Quan, Ruiming Tang, Yong Yu, and Weinan Zhang. 2024. Rella: Retrieval-enhanced large language models for lifelong sequential behavior comprehension in recommendation. In *Proceedings of the ACM Web Conference 2024*. 3497–3508.

[20] Xiaoqiang Lin, Aritra Ghosh, Bryan Kian Hsiang Low, Anshumali Shrivastava, and Vijai Mohan. 2025. Refrag: Rethinking rag based decoding. *arXiv preprint arXiv:2509.01092* (2025).

[21] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual instruction tuning. *Advances in neural information processing systems* 36 (2023), 34892–34916.

[22] Xueguang Ma, Liang Wang, Nan Yang, Furu Wei, and Jimmy Lin. 2024. Fine-tuning llama for multi-stage text retrieval. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2421–2425.

[23] Chuan Meng, Negar Arabzadeh, Arian Askari, Mohammad Aliannejadi, and Maarten de Rijke. 2025. Query performance prediction using relevance judgments generated by large language models. *ACM Transactions on Information Systems* 43, 4 (2025), 1–35.

[24] Ronak Pradeep, Sahel Sharifymoghaddam, and Jimmy Lin. 2023. Rankvicuna: Zero-shot listwise document reranking with open-source large language models. *arXiv preprint arXiv:2309.15088* (2023).

[25] Zhen Qin, Rolf Jagerman, Kai Hui, Honglei Zhuang, Junru Wu, Le Yan, Jiaming Shen, Tianqi Liu, Jialu Liu, Donald Metzler, et al. 2024. Large language models are effective text rankers with pairwise ranking prompting. In *Findings of the Association for Computational Linguistics: NAACL 2024*. 1504–1518.

[26] Zhen Qin, Rolf Jagerman, Rama Kumar Pasumarthi, Honglei Zhuang, He Zhang, Aijun Bai, Kai Hui, Le Yan, and Xuanhui Wang. 2023. Rd-suite: A benchmark for ranking distillation. *Advances in Neural Information Processing Systems* 36 (2023), 35748–35760.

[27] Sashank Reddi, Rama Kumar Pasumarthi, Aditya Menon, Ankit Singh Rawat, Felix Yu, Seungyeon Kim, Andreas Veit, and Sanjiv Kumar. 2021. Rankdistil: Knowledge distillation for ranking. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 2368–2376.

[28] Devendra Sachan, Mike Lewis, Mandar Joshi, Armen Aghajanyan, Wen-tau Yih, Joelle Pineau, and Luke Zettlemoyer. 2022. Improving Passage Retrieval with Zero-Shot Question Generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 3781–3797.

[29] Hongwei Shang, Nguyen Vo, Nitin Yadav, Tian Zhang, Ajit Puthenputhussery, Xunfan Cai, Shuyi Chen, Prijith Chandran, and Changsung Kang. 2025. Knowledge Distillation for Enhancing Walmart E-commerce Search Relevance Using Large Language Models. In *Companion Proceedings of the ACM on Web Conference 2025*. 449–457.

[30] Sundara Raman Ramachandran. 2025. [Generative Score API] Multi-Item scoring with custom attention mask (Pull request #10979). https://github.com/sgl-project/sglang/pull/10979. Pull request #10979 in the SGLang project.

[31] Raphael Tang, Yao Lu, Linqing Liu, Lili Mou, Olga Vechtomova, and Jimmy Lin. 2019. Distilling task-specific knowledge from bert into simple neural networks. *arXiv preprint arXiv:1903.12136* (2019).

[32] Han Wang, Mukuntha Narayanan Sundararaman, Onur Gungor, Yu Xu, Krishna Kamath, Rakesh Chalasani, Kurchi Subhra Hazra, and Jinfeng Rao. 2024. Improving Pinterest Search Relevance Using Large Language Models. *arXiv preprint arXiv:2410.17152* (2024).

[33] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[34] Likang Wu, Zhi Zheng, Zhaopeng Qiu, Hao Wang, Hongchao Gu, Tingjia Shen, Chuan Qin, Chen Zhu, Hengshu Zhu, Qi Liu, et al. 2024. A survey on large language models for recommendation. *World Wide Web* 27, 5 (2024), 60.

[35] Xuansheng Wu, Huachi Zhou, Yucheng Shi, Wenlin Yao, Xiao Huang, and Ninghao Liu. 2024. Could small language models serve as recommenders? towards data-centric cold-start recommendation. In *Proceedings of the ACM Web Conference 2024*. 3566–3575.

[36] Runze Xia, Yupeng Ji, Yuxi Zhou, Haodong Liu, Teng Zhang, and Piji Li. 2025. From Reasoning LLMs to BERT: A Two-Stage Distillation Framework for Search Relevance. *arXiv preprint arXiv:2510.11056* (2025).

[37] Dezhi Ye, Jie Liu, Junwei Hu, Jiabin Fan, Bowen Tian, Haijin Liang, and Jin Ma. 2025. Applying Large Language Model For Relevance Search In Tencent. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*. 5171–5181.

[38] Junjie Zhang, Ruobing Xie, Yupeng Hou, Xin Zhao, Leyu Lin, and Ji-Rong Wen. 2025. Recommendation as instruction following: A large language model empowered recommendation approach. *ACM Transactions on Information Systems* 43, 5 (2025), 1–37.

[39] Longhui Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, Meishan Zhang, and Min Zhang. 2024. A Two-Stage Adaptation of Large Language Models for Text Ranking. In *ACL (Findings)*.

[40] Yang Zhang, Fuli Feng, Jizhi Zhang, Keqin Bao, Qifan Wang, and Xiangnan He. 2025. Collm: Integrating collaborative embeddings into large language models for recommendation. *IEEE Transactions on Knowledge and Data Engineering* (2025).

[41] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3848–3860. doi:10.14778/3611540.3611569

[42] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

[43] Shengyao Zhuang, Honglei Zhuang, Bevan Koopman, and Guido Zuccon. 2024. A setwise approach for effective and highly efficient zero-shot ranking with large language models. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 38–47.