# Evaluating Graph Transformers for Bluesky Recommendations

**Aleck Wu** a5wu@ucsd.edu
**SangGyu An** sgan@ucsd.edu
**Yusu Wang** yusuwang@ucsd.edu
**Gal Mishne** gmishne@ucsd.edu

**Abstract**

Bluesky is a decentralized social media platform similar to Twitter, where the data on the network is open for access to everyone. Compared to other recommendation datasets like Last.fm or MovieLens, social media interactions can evolve rapidly, requiring models to adapt to changing user preferences over time. Traditional collaborative filtering requires matrix factorization on the entire user-item rating/interaction matrix, requiring periodic recomputation. This may work well on music or movie recommendations, but for a social media site like Bluesky, a scalable way to compute and update embeddings for in real-time is needed. We demonstrate a simple, scalable framework for computing embeddings for any social network, and also show a Graph Transformer-based re-ranking model that refines rankings using both graph structure and temporal information.

Website: https://bbeat2782.github.io/Bluesky-GraphRec/
Code: https://github.com/bbeat2782/Bluesky-GraphRec

# 1 Introduction

Decentralized social media platforms like Bluesky represent a new paradigm in online social interaction, characterized by open data access and user autonomy. Unlike centralized platforms, the transparent nature of Bluesky offers unique opportunities for researchers and developers to build novel applications and services, including personalized recommendation systems. However, the dynamic and rapidly evolving nature of social interactions on such platforms poses significant challenges for traditional recommendation techniques. User preferences on social media are often fleeting and influenced by real-time trends, requiring recommendation models to be highly adaptive and responsive to temporal shifts in user behavior.

Broadly, recommendation systems fall into two dominant categories: content-based and graph-based models. Content-based models, such as two-tower architectures, rely on separate encoders to process explicit user and item features (e.g., user profiles, item descriptions). These representations are then projected into a shared latent space, often through contrastive learning, to facilitate recommendations based on feature similarity. In contrast, graph-based models, most notably collaborative filtering (CF) methods like matrix factorization, derive user and item embeddings directly from the network of interactions. CF's strength lies in its simplicity and power; it learns representations implicitly from the graph structure itself, bypassing the need for pre-defined explicit features. For a platform like Bluesky, where user connections and interactions within the network graph constitute the primary signal, CF emerges as a particularly compelling starting point. Explicit features, such as detailed user profiles or rich post content metadata (beyond basic text), are comparatively less structured and readily available in this decentralized social media context, making graph structure a more robust and reliable foundation for recommendations.

Matrix factorization based methods have been successful in various recommendation domains such as music and movie recommendations. These methods typically rely on static user-item interaction matrices, necessitating periodic re-computation of embeddings to account for new interactions. While effective for domains with relatively stable preferences, this approach becomes computationally expensive and less suitable for dynamic social media environments like Bluesky, where interactions occur at a high velocity and user interests can shift significantly even within hours.

To address these challenges, we propose a two-stage, scalable framework for dynamic post recommendation on Bluesky. Our framework is designed to efficiently generate and update user and post embeddings in a real-time manner, and subsequently refine recommendations using a graph transformer-based re-ranking model that effectively captures both graph structural information and temporal dynamics of user-post interactions.

Our GraphRec Graph Transformer model builds on top of DyGFormer[9] for dynamic graph learning. The rationale for leveraging a Graph Transformer architecture stems from its ability to model complex temporal dependencies and capture nuanced patterns within dynamic interaction sequences.

# 2 Methods

## 2.1 Data Collection and Preprocessing

We collect data for 6 months from the start of the Bluesky network from the Bluesky PDSes. We collect posts, likes, and follows.

## 2.2 Candidate Generation Framework

User-Item based Collaborative filtering leverages patterns of user-item interactions to infer preferences. It assumes that users that interact similarly in the past will interact similarly in the future. The goal is to approximately reconstruct the full interaction matrix $I$ as a product of two lower-dimensional matrices:

$$I \approx UV^T$$

with:

$$\min_{U,V} ||I - UV^T||^2$$

minimizing the reconstruction error where:

- $U \in \mathbb{R}^{|U| \times d}$ represents latent user factors,
- $V \in \mathbb{R}^{|V| \times d}$ represents latent item factors,
- $d$ is the dimensionality of the latent space.

As mentioned before, directly computing $I = UV^T$ is expensive, and because of the real-time nature of the data, it is intractable to compute it each time there is a new interaction. Instead, we need a way to incrementally update one of the latent factor matrices.

The framework to get user and post embeddings most similar to our approach would be Twitter's SimClusters [4], but whereas SimClusters is more complicated and involves many different steps, some of which are done in distributed fashion, ours is simpler and intuitive to understand, involving just two steps that can be done on a single machine.

Instead of using the full interaction graph to compute both the user embeddings and the post embeddings, we only need to get the user embeddings first by factorizing the much smaller user-user follow matrix:

$$F = CP^T$$

where:

- $C \in \mathbb{R}^{|C| \times d}$ represents the consumer embeddings.
- $P \in \mathbb{R}^{|P| \times d}$ represents the producer embeddings.
- $d$ is the dimensionality of the latent space.

Here we define consumers to mean users that follow others users. We define producers to mean users that get followed by other users. By factorizing this matrix, which is orders of magnitudes smaller, we can efficiently get meaningful user embeddings. Since the majority of users follow and consume content made by a few producers, we can further cut down on the user-user follow graph by only using small portion of the users on the right side of the bipartite

graph (those with >30 followers). And since follows don't happen as often as post interactions, we can safely compute this in batch periodically.

For posts, each post $V_j^{(0)}$ is initialized with the producer vector. Then we have the following update rule: When a user $k$ interacts with post $j$ (e.g. by liking it), the post embedding is updated as:

$$V_j \leftarrow \frac{V_j + C_k}{||V_j + C_k||_2}$$

Now we have found a way to update the post embeddings incrementally, a requirement for real-time recommendation. This online update process, drawing an analogy to Graph Neural Networks, can be seen as a simplified form of message passing. User interactions act as dynamic signals, with each consumer liking a post effectively contributing their user embedding ($C_k$) to refine the post's representation. The post embedding then aggregates these signals through summation and normalization, evolving over time in response to user engagement. Importantly, this was done purely by learning from the graph structure, without defining any explicit features.

It is also important to note that we can only derive user embeddings due to the "high signal" nature of follow relationships on Bluesky. It is precisely because the user embeddings are meaningful that we can compute post embeddings as well. To adapt this framework to other social media sites, you would need to keep in mind the equivalent high-signal user connections specific to each platform. On Reddit, for example, you would replace the consumer-producer follow graph with a user-subreddit subscribe graph in order to get the user embeddings.

We use these user embeddings and dynamic temporal post embeddings for both candidate generation and to train our second-stage GraphRec model.

## 2.3 On Metrics

For developers to build an effective recommendation system on the Bluesky network, there must be high signal inherent in the data. The decentralized AT Protocol on Bluesky has the following high signal data points: likes, follows, and posts. Developers must rely on offline evaluation metrics, since they do not have access to most users. As such, this puts them at a disadvantage compared to operators like Bluesky that can evaluate models with online metrics such as A/B testing on users that are using their algorithm.

There is inherent irreducible bias in offline evaluation. Recommender systems do not predict what users will do; rather, they offer choices to the users that themselves affect user actions. This is excacerbated by the fact that users follow a different algorithm, namely Bluesky's. Thus, maximizing hit rate for recommending posts would be somewhat correlated to maximizing closeness to Bluesky's "For You" algorithm.

Ultimately, online A/B tests are the gold standard for testing recommendation systems. Offline metrics can only serve as proxies and might lead to conclusions that don't hold up in production. That being said, if we look at the nature of likes on Bluesky, users may use likes to encourage the algorithm to show them more of similar content. Compared to watch time, likes are likely

a strong indication that a user has a positive affinity for a post.

To evaluate the performance of our pipeline, we use Hit@k, MRR, and ILD.

## 2.4 GraphRec Model Implementation

After reducing the computational complexity by limiting the number of neighbors considered per node, we implement the GraphRec model for link prediction. The model extracts 2-hops neighbors and their features, then applies transformers to capture long-term temporal dependencies in user-post interactions.

**Algorithm 1:** GraphRec Model Implementation

**Input:** $\mathcal{G} = (\mathcal{U} \cup \mathcal{P}, \mathcal{E})$, where $\mathcal{G}$ represents a graph, $\mathcal{U}$ and $\mathcal{P}$ are user and post nodes, and $\mathcal{E}$ represents edges.
**Output:** $\hat{y}$, where $\hat{y}$ represents link prediction score.
**Step 1: Sampling Past Interactions:**

$$\mathcal{N}_{\text{recent}}(v_i, t_i) = \text{Top-}k(\mathcal{N}(v_i, t_i), \text{key} = t_j)$$

*Explanation:* For a given node $v_i$, sample its most recent $k$ neighbors from its interaction history before a specific time $t_i$.
**Step 2: Features:** Node features $\mathbf{X}_{node,i}$ are preprocessed and temporal features $\mathbf{X}_{time,t}$ are encoded:

$$\mathbf{X}_t = \sqrt{\frac{1}{d_T}}[\cos(w_1 \Delta t'), \sin(w_1 \Delta t'), \ldots, \cos(w_{d_T} \Delta t'), \sin(w_{d_T} \Delta t')]$$

where $\Delta t' = t - t'$ and $d_T$ is the encoding dimension.
Neighbor co-occurrence features are encoded:

$$\mathbf{H}_{co}(v_i) = \sum_{j \in \mathcal{N}(v_i)} \sigma\left(W_2 \cdot \text{ReLU}\left(W_1 \sum_{n \in \mathcal{N}(v_i)} \mathbb{1}(n \in \mathcal{N}(v_j)) + b_1\right) + b_2\right)$$

where $\mathcal{N}(v_i)$ denotes the set of neighbors of node $v_i$, $\mathbb{1}$ is an indicator function, $W_1, W_2 \in \mathbb{R}^{d \times d}$ are learnable weight matrices, and $b_1, b_2 \in \mathbb{R}^d$ are learnable bias terms.
**Step 3: Patching:** Divide sequences into patches:

$$\mathbf{X} \to \mathbf{X}_{\text{patched}} \in \mathbb{R}^{\frac{L_{\max}}{p} \times (p \cdot d)}$$

where $L_{\max}$ is max input sequence length, $p$ is patch size, and $d$ is feature dimension.
**Step 4: Projection:** Project features to shared space:

$$\mathbf{X}_{\text{proj}} = \mathbf{X}_{\text{patched}} \cdot \mathbf{W}$$

where $\mathbf{W} \in \mathbb{R}^{p \cdot d \times d'}$ with $d'$ as project dimension.
**Step 5: Concatenation:** Combine source and destination node features:

$$\mathbf{X} = [\mathbf{X}_{\text{src\_proj}}; \mathbf{X}_{\text{dst\_proj}}] \in \mathbb{R}^{2 \cdot \text{num patches} \times d'}$$

where num patches $= \frac{L_{\max}}{p}$.
**Step 6: Transformer Encoding:** Apply self-attention to learn dependencies within and across sequences:

$$\mathbf{Q} = \mathbf{X}_{\text{norm}} \mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}_{\text{norm}} \mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}_{\text{norm}} \mathbf{W}_V$$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

$$\mathbf{X}_{\text{attn}} = \mathbf{X} + \text{Dropout}(\text{MultiHeadAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}))$$

$$\mathbf{X}_{\text{out}} = \mathbf{X}_{\text{attn}} + \text{Dropout}(\text{FFN}(\mathbf{X}_{\text{attn}}))$$

where:

- $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d' \times d_k}$ are projection matrices.
- $\mathbf{X}_{\text{out}} \in \mathbb{R}^{2 \cdot \text{num patches} \times d'}$.

**Step 7: Node Embeddings:** Compute source and destination embeddings:

$$\mathbf{H} = \text{Mean}(\mathbf{X}) \cdot \mathbf{W}_{\text{out}} + \mathbf{b}_{\text{out}} \in \mathbb{R}^{d_{\text{out}}}$$

where taking the mean is edge-level pooling, $\mathbf{W}_{\text{out}} \in \mathbb{R}^{d' \times d_{\text{out}}}$, and $\mathbf{b}_{\text{out}} \in \mathbb{R}^{d_{\text{out}}}$.
**Step 8: Link Prediction:**

$$\hat{y} = \text{ReLU}([\mathbf{H}_{\text{src}}; \mathbf{H}_{\text{dst}}] \cdot \mathbf{W}_1 + \mathbf{b}_1) \cdot \mathbf{W}_2 + \mathbf{b}_2$$

where $\mathbf{W}_1 \in \mathbb{R}^{2d_{\text{node}} \times d_{\text{hidden}}}$, $\mathbf{b}_1 \in \mathbb{R}^{d_{\text{hidden}}}$, $\mathbf{W}_2 \in \mathbb{R}^{d_{\text{hidden}} \times 1}$, and $\mathbf{b}_2 \in \mathbb{R}^1$.

### 2.4.1 Sampling Past Interactions

For a given node $v_i$, sample its 1-hop neighbors based on the most recent interactions that occurred before a specific interaction time $t_i$. The neighbors are defined as:

$$\mathcal{N}(v_i, t_i) = \{v_j \mid (v_i, v_j, t_j) \in \mathcal{E},\ t_j < t_i\}, \quad |\mathcal{N}(v_i, t_i)| = \text{max input sequence length}$$

To ensure consistency, only the most recent interactions are retained:

$$\mathcal{N}_{\text{recent}}(v_i, t_i) = \text{Top-}k\big(\mathcal{N}(v_i, t_i), \text{key} = t_j, \text{descending}\big)$$

where:

- $\mathcal{N}(v_i, t_i)$ is the set of neighbors for node $v_i$ at time $t_i$, filtered to include only interactions before $t_i$.
- Top-$k$ selects the $k = \text{max input sequence length}$ neighbors with the most recent interaction times $t_j$.
- key $= t_j$ ensures neighbors are sorted by their interaction times $t_j$ in descending order.

### 2.4.2 Features

- User-node features: $\mathbf{U}_i \in \mathbb{R}^{64}$, obtained via low-rank matrix factorization using truncated SVD on the user-producer interaction matrix. These embeddings capture latent user preferences and are L2-normalized for similarity-based retrieval.
- Post-node features: $\mathbf{P}_i \in \mathbb{R}^{192}$, with text embeddings generated by Jina Embeddings v3 [5], concatenated with user-based post embeddings.
- Temporal features (interaction times): Temporal differences between interactions are encoded using a method described by [7], where relative time intervals rather than absolute timestamps are used to capture meaningful patterns in interactions. The goal is to learn a continuous functional mapping:

  $$\Phi : T \to \mathbb{R}^{d_T}$$

  which maps time intervals to a $d_T$-dimensional vector space, serving as a replacement for positional encoding in self-attention. The encoding is defined as:

  $$\mathbf{X}^t_{U_i,P_i} = \sqrt{\frac{1}{d_T}}\left[\cos(w_1\Delta t'), \sin(w_1\Delta t'), \ldots, \cos(w_{d_T}\Delta t'), \sin(w_{d_T}\Delta t')\right]$$

  where:
  - $\Delta t' = |t - t'|$ is the relative time difference between two interactions.
  - $d_T$ is the encoding dimension.
  - $w_1, w_2, \ldots, w_{d_T}$ are trainable parameters optimized jointly with the model.

  This encoding method is compatible with self-attention and can effectively replace traditional positional encodings to learn interaction-based temporal patterns.

- Neighbor co-occurrence features: These features capture shared nodes between users and posts by computing the frequency with which a given node appears in the neighborhoods of others. The neighbor co-occurrence matrix is defined as:

$$C(v_i, v_j) = \sum_{n \in \mathcal{N}(v_i)} \mathbb{1}(n \in \mathcal{N}(v_j))$$

where $\mathcal{N}(v_i)$ represents the set of neighbors of node $v_i$, and $\mathbb{1}(\cdot)$ is an indicator function that counts occurrences. These co-occurrence counts are transformed using a non-linear projection:

$$\mathbf{H}_{co}(v_i) = \sum_{j \in \mathcal{N}(v_i)} \sigma\left(W_2 \cdot \mathrm{ReLU}\left(W_1 C(v_i, \mathcal{N}(v_i)) + b_1\right) + b_2\right)$$

where $W_1, W_2$ are learnable projection matrices, $b_1, b_2$ are bias terms, and $\sigma(\cdot)$ is an activation function such as sigmoid or softmax. This step ensures that the model learns structural similarities between nodes in the interaction graph.
To incorporate broader structural context, 1-hop and 2-hop neighbors are concatenated before computing co-occurrence counts, allowing the model to capture both direct and indirect interactions without differentiating them.

### 2.4.3 Patching

To capture long-term temporal dependencies while maintaining computational efficiency, a patching technique inspired by [9] is adopted. This approach splits each sequence into multiple patches, which are then fed into a Transformer. The patching strategy effectively preserves local temporal proximities and enables the model to efficiently process longer histories.

The idea of dividing the input sequence into patches has been applied in various domains. For instance:

- Vision Transformer (ViT) [1] splits an image into multiple patches and feeds the sequence of linear embeddings of these patches into a Transformer, achieving remarkable performance in image classification tasks.
- PatchTST [2] divides time series data into subseries-level patches, which are processed by a channel-independent Transformer for long-term multivariate time series forecasting.

In this work, a similar patching mechanism is applied to sequences of node and temporal features. Specifically:

- A node features, initially of shape [max input sequence length, feature dimension size], are divided into patches depending on patch size $p$.

$$\mathbf{X} \in \mathbb{R}^{\text{max input sequence length} \times \text{feature dim}} \quad \rightarrow \quad \mathbf{X}_{\text{patched}} \in \mathbb{R}^{\frac{\text{max input sequence length}}{p} \times p \cdot \text{feature dim}}$$

By splitting the input into patches, the Transformer processes sub-sequences instead of the entire sequence, reducing the computational complexity typical of self-attention mechanisms.

This approach not only aligns with prior studies on leveraging patch-based input transformations but also ensures that the temporal relationships and structural proximities within the sequence are effectively preserved, enhancing the model's ability to learn from long histories.

### 2.4.4 Projection

After dividing the input sequences into patches, feature projection is applied to align the patch encoding dimensions. This is achieved using separate linear layers for node features and temporal features.

The projection layers are:

$$\text{Projection Layer} = \begin{cases} \mathbf{W}_{\text{node}} \in \mathbb{R}^{p \cdot \text{node feat dim} \times d'} \\ \mathbf{W}_{\text{time}} \in \mathbb{R}^{p \cdot \text{time feat dim} \times d'} \end{cases}$$

where $d'$ is a project dimension. The features for source and destination nodes are transformed as:

$$\mathbf{X}_{\text{node\_proj}} = \mathbf{X}_{\text{node\_patched}} \cdot \mathbf{W}_{\text{node}}, \quad \mathbf{X}_{\text{time\_proj}} = \mathbf{X}_{\text{time\_patched}} \cdot \mathbf{W}_{\text{time}}$$

where:

- $\mathbf{X}_{\text{node\_patched}} \in \mathbb{R}^{\frac{\text{max input sequence length}}{p} \times p \cdot \text{feature dim}}$ is the patched node feature matrix.
- $\mathbf{X}_{\text{node\_proj}} \in \mathbb{R}^{\frac{\text{max input sequence length}}{p} \times d'}$ is the projected node feature matrix.
- $\mathbf{W}_{\text{node}}$ and $\mathbf{W}_{\text{time}}$ are learnable parameters.

This feature projection aligns the dimensions of all patches, enabling them to be concatenated and passed into the Transformer layers. It also ensures that both node and temporal information are represented in a unified embedding space.

### 2.4.5 Transformer Encoding

After feature projection on the input data to have the same dimension, concatenation is applied for each source and destination node.

$$\mathbf{X}_{\text{concat}} = \mathbf{X}_{\text{node\_proj}} || \mathbf{X}_{\text{time\_proj}} \in \mathbb{R}^{\frac{\text{max input sequence length}}{p} \times 2d'}$$

Then, instead of processing source and destination nodes individually, we stacked them before inputting them so that the model can learn the temporal dependencies within and across the sequences. Within-sequence dependencies model how features evolve over time for a specific node type (e.g., user-to-post interactions or post-to-user interactions separately). This captures how a user's interaction history progresses over time or how a post gains interactions. Across-sequence dependencies capture how user and post interactions influence each other over time, helping the model understand cross-sequence relationships between users and posts.

$$\mathbf{X} = [\mathbf{X}_{\text{src\_concat}}; \mathbf{X}_{\text{dst\_concat}}] \in \mathbb{R}^{2 \cdot \frac{\text{max input sequence length}}{p} \times 2d'}$$

Once the concatenated and stacked features are prepared, they are passed through Transformer encoder layers. The Transformer encoder, introduced by [6], consists of the following key components:

- **Multi-Head Attention:** The model computes self-attention for each sequence of patches to capture temporal and structural dependencies:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

  where:
  - $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{2 \cdot \text{num patches} \times d}$ are the query, key, and value matrices derived from the input, where $d$ is the embedding dimension of each token and $\frac{\text{max input sequence length}}{p} =$ num patches.
  - $d_k$ is the dimensionality of the key vectors.
- **Feed-Forward Network (FFN):** After the attention layer, the output is passed through a two-layer feed-forward network:

$$\text{FFN}(\mathbf{X}) = \text{GELU}(\mathbf{X}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

  where $\mathbf{W}_1, \mathbf{W}_2$ are trainable weight matrices.
- **Residual Connections and Layer Normalization:** To ensure stable learning, residual connections are applied, followed by layer normalization:

$$\mathbf{X}_{\text{out}} = \text{LayerNorm}(\mathbf{X}_{\text{in}} + \text{Dropout}(\text{Attention/FFN Output}))$$

The entire steps for processing the input tensor $\mathbf{X} \in \mathbb{R}^{2 \cdot \text{num patches} \times 2d'}$ is the following:

$$\mathbf{X}_{\text{norm}} = \text{LayerNorm}(\mathbf{X})$$

$$\mathbf{Q} = \mathbf{X}_{\text{norm}}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}_{\text{norm}}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}_{\text{norm}}\mathbf{W}_V$$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

$$\mathbf{X}_{\text{attn}} = \mathbf{X} + \text{Dropout}(\text{MultiHeadAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}))$$

$$\mathbf{X}_{\text{ffn}} = \text{GELU}(\mathbf{X}_{\text{attn}}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

$$\mathbf{X}_{\text{out}} = \mathbf{X}_{\text{attn}} + \text{Dropout}(\mathbf{X}_{\text{ffn}})$$

$$\mathbf{X}_{\text{out}} \in \mathbb{R}^{2 \cdot \text{num patches} \times 2d'}$$

where:

- $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{2d' \times d}$ are trainable weight matrices for computing queries, keys, and values.
- $\mathbf{W}_1 \in \mathbb{R}^{2d' \times d_{\text{hidden}}}, \mathbf{W}_2 \in \mathbb{R}^{d_{\text{hidden}} \times 2d'}$ are FFN weight matrices.
- $\mathbf{b}_1, \mathbf{b}_2$ are bias terms.
- Dropout$(\cdot)$ applies dropout for regularization.
- GELU$(\cdot)$ is the Gaussian Error Linear Unit activation function.
- $d'$ is the project dimension from step 4.

### 2.4.6 Node Embeddings

After passing through the Transformer encoder layers, the processed tensor is split back into source and destination sequences:

$$\mathbf{X}_{\text{src}} = \mathbf{X}[: \text{num patches}, :], \quad \mathbf{X}_{\text{dst}} = \mathbf{X}[\text{num patches} :, :]$$

The final node embeddings for source and destination nodes are derived by averaging their patch representations and applying a fully connected output layer:

$$\mathbf{H}_{\text{src}} = \text{Mean}(\mathbf{X}_{\text{src}})\mathbf{W}_{\text{out}} + \mathbf{b}_{\text{out}} \in \mathbf{R}^{d_{\text{out}}}, \quad \mathbf{H}_{\text{dst}} = \text{Mean}(\mathbf{X}_{\text{dst}})\mathbf{W}_{\text{out}} + \mathbf{b}_{\text{out}} \in \mathbf{R}^{d_{\text{out}}}$$

where:

- $\mathbf{H}_{\text{src}}, \mathbf{H}_{\text{dst}} \in \mathbb{R}^{\text{node feat dim}}$ are the final node embeddings.
- $\mathbf{W}_{\text{out}} \in \mathbb{R}^{2 \cdot d' \times \text{node feat dim}}$ is the trainable weight matrix.
- $\mathbf{b}_{\text{out}} \in \mathbb{R}^{\text{node feat dim}}$ is the trainable bias vector.

### 2.4.7 Link Prediction

The link predictor is implemented by concatenating the source and destination node embeddings and projects them to the desired output dimension. The process involves the following steps:

- **Input Concatenation:** The source ($\mathbf{H}_{\text{src}}$) and destination ($\mathbf{H}_{\text{dst}}$) node embeddings are concatenated along the feature dimension:

  $$\mathbf{H}_{\text{concat}} = [\mathbf{H}_{\text{src}}; \mathbf{H}_{\text{dst}}] \in \mathbb{R}^{2d_{\text{node}}}$$

  where $d_{\text{node}}$ is the dimension of the node embeddings.
- **Hidden Layer Transformation:** The concatenated features are passed through a hidden layer with ReLU activation:

  $$\mathbf{H}_{\text{hidden}} = \text{ReLU}(\mathbf{H}_{\text{concat}}\mathbf{W}_1 + \mathbf{b}_1) \in \mathbb{R}^{d_{\text{hidden}}}$$

  where $\mathbf{W}_1 \in \mathbb{R}^{2d_{\text{node}} \times d_{\text{hidden}}}$ and $\mathbf{b}_1 \in \mathbb{R}^{d_{\text{hidden}}}$ are trainable parameters.
- **Output Projection:** The hidden layer output is projected to a scalar value indicating the likelihood of a link:

  $$\hat{y} = \mathbf{H}_{\text{hidden}}\mathbf{W}_2 + \mathbf{b}_2 \in \mathbb{R}$$

  where $\mathbf{W}_2 \in \mathbb{R}^{d_{\text{hidden}} \times 1}$ and $\mathbf{b}_2 \in \mathbb{R}$ are trainable parameters.

The link predictor learns to estimate the score of a link existing between the source and destination nodes based on their embeddings. This score is then evaluated using Bayesian Personalized Ranking (BPR) loss.

## 2.5 Training

For each positive interaction in the training set, we randomly sample four negative posts from the last 20-minute timeframe to construct negative interactions. The model is trained using Bayesian Personalized Ranking (BPR) loss, optimizing for pairwise ranking. We evaluate model performance using accuracy and pairwise accuracy, ensuring that positive interactions rank higher than negative ones. Mean Reciprocal Rank (MRR) is not used to reduce training time, as computing the full ranking position for each sample is computationally expensive.

### 2.5.1 Bayesian Personalized Ranking (BPR) Loss

The BPR loss is designed to optimize ranking by enforcing that positive interactions are ranked higher than sampled negative interactions [3]. The loss function is defined as:

$$\mathscr{L}_{BPR} = -\frac{1}{|D|} \sum_{(u,p^+,p^-)\in D} \log \sigma(\hat{y}_{up^+} - \hat{y}_{up^-}) + \lambda ||\Theta||^2$$

where:

- $(u, p^+, p^-)$ represents a triplet containing user $u$, a positive post $p^+$ that the user interacted with, and a randomly sampled negative post $p^-$.
- $\hat{y}_{up^+}$ and $\hat{y}_{up^-}$ are the predicted scores for positive and negative interactions.
- $\sigma(x)$ is the sigmoid function.
- $\lambda ||\Theta||^2$ is the regularization term to prevent overfitting.
- $D$ is the total number of positive interactions in the dataset.

### 2.5.2 Evaluation Metrics

To measure performance, we use:

- **Accuracy@1**: Measures how often the positive is ranked highest.
- **Pairwise Accuracy**: Measures how often the positive interaction is ranked higher than each of the negative interactions:

$$\text{Pairwise Acc} = \frac{1}{|D|} \sum_{(u,p^+,\{p_1^-,\ldots,p_N^-\})\in D} \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}(\hat{y}_{up^+} > \hat{y}_{up_i^-})$$

  where:
  - $N = 4$ (number of negative samples per positive sample).
  - $\mathbb{1}(\cdot)$ is an indicator function that returns 1 if the condition is met, otherwise 0.

### 2.5.3 Training and Inference

The training process consists of 50 epochs with early stopping if there is no improvement for five consecutive epochs. Training is currently performed on 1,042,739 interactions. After training, the inference step generates recommendations by predicting the most relevant items for each user. The inference phase is conducted on 14,073 users to predict the last post each user interacted with between 2023-06-14 and 2023-06-23. Each user is provided with 2,000 candidate posts, which are selected through the candidate generation step.

## 2.6 Comparisons

### 2.6.1 Popularity-Based Recommendation

The popularity-based approach recommends Bluesky posts purely based on the number of likes a post has received. The assumption is that highly liked posts are more engaging and relevant to users and users are influenced by social proof, engaging with posts that others have already liked. While this method is simple and computationally efficient, it has several limitations:

- **Lack of Personalization**: All users receive similar recommendations, disregarding individual preferences.
- **Cold Start Problem**: New posts with few or no likes struggle to gain visibility.
- **Temporal Bias**: Older posts with accumulated likes are favored over newer content, leading to potential stagnation in recommendations.

### 2.6.2 MLP-based Neighborhood Aggregation

Unlike the popularity-based approach, which solely relies on the number of likes, the MLP-based neighborhood aggregation method learns user preferences by leveraging temporal information and interactions. This method utilizes Multi-Layer Perceptrons (MLPs) to aggregate features from a user's local network, dynamically adapting to changing behaviors over time.

The architecture follows a temporal graph representation, where each node (representing a user or post) updates its embedding based on historical interactions. Instead of relying on multi-head attention mechanisms, this approach aggregates neighborhood information using mean pooling and MLP layers.

This approach improves upon the limitations of popularity-based methods by incorporating temporal dependencies, user-specific behavior, and relational information, resulting in a more adaptive and context-aware recommendation system.

The backbone of this model is based on [8], where the multi-head attention mechanism has been replaced with an MLP to establish a baseline.

### 2.6.3 Configurations

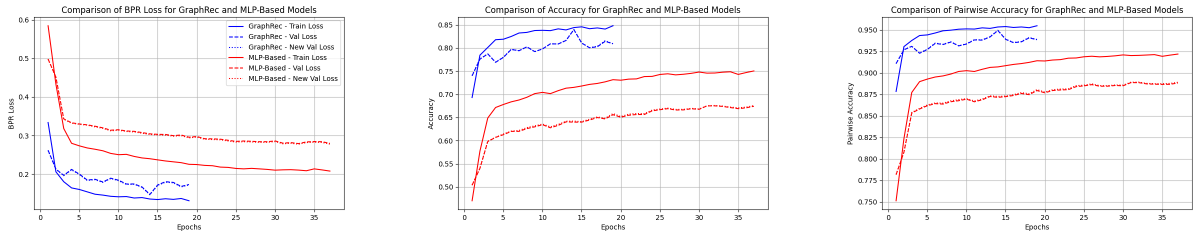
To ensure a fair comparison, we maintain consistent hyperparameters across all models. Each model utilizes a 2-hop neighborhood with 10 neighbors per hop, resulting in a receptive field of $10 + 10 \times 10$ nodes. The time feature dimension is set to 100, while the channel embedding dimension is 50, representing the node embedding size before applying a transformer or MLP. Training is conducted using an Adam optimizer with a learning rate of 0.0001, a dropout rate of 0.1, and no weight decay. Each model is trained for a maximum of 50 epochs, with early stopping triggered after 5 epochs of no improvement on both evaluation metrics, ensuring efficient convergence. The process is repeated 3 times, and all experiments are conducted on a single NVIDIA A5000 GPU.

In addition to these shared hyperparameters, GraphRec incorporates 2 attention heads and a patch size of 5, allowing it to capture more refined structural patterns in the graph data.

# 3 Results

For candidate generation, the metrics are calculated by simulating the real-time recommendation through a forward-time process. We treat every point of interaction as an opportunity for recommendation. We accumulate the hit rate as well as update the post embeddings after each recommendation event.

## 3.1 Training Result




(a) BPR Loss (b) Accuracy (c) Pairwise accuracy

Figure 1: BPR Loss, accuracy, and pairwise accuracy for training GraphRec and MLP-Based models

Figure 1 presents a comparison of the training performance between the GraphRec and MLP-Based models across BPR Loss, Accuracy, and Pairwise Accuracy. Overall, GraphRec demonstrates superior performance in all three metrics, suggesting that it effectively captures graph-based interactions compared to the MLP-Based model.

New Val is a subset of the validation set, comprising only interactions in which either the user or the post node was unseen during training. Within each model, the differences between the

validation and new node validation results are relatively small, indicating that both models generalize well to unseen interactions.

Between the two models, GraphRec consistently achieves a lower BPR loss than the MLP-Based model, as shown in Figure 1a, indicating more effective optimization. While both models exhibit a decreasing loss trend, GraphRec stabilizes at a lower value, whereas the MLP-Based model demonstrates a slower convergence rate.

In terms of accuracy (Figure 1b), GraphRec significantly outperforms the MLP-Based model across the training, validation, and new node validation sets. The MLP-Based model improves steadily and takes longer to trigger early stopping. Similarly, as seen in Figure 1c, pairwise accuracy trends show that GraphRec achieves higher ranking performance earlier, while the MLP-Based model requires more training epochs to reach comparable improvements.

## 3.2 Inference Results

In addition to MRR and Hit Rate, we also evaluate Intra-List Diversity (ILD) to assess the diversity of recommendations.

### 3.2.1 Intra-List Diversity (ILD)

Most recommendation models prioritize ranking accuracy, such as Mean Reciprocal Rank (MRR), which optimizes the placement of relevant items. However, focusing solely on accuracy can lead to redundancy in recommendations, where users receive items that are too similar to each other. This reduces content diversity and limits exposure to new or less popular items. To address this, we evaluate Intra-List Diversity (ILD), which quantifies how varied the recommendations are within a user's top-ranked results.

Intra-List Diversity (ILD) measures the pairwise dissimilarity among items in a user's recommendation list. It is computed as:

$$ILD = 1 - \frac{1}{N(N-1)} \sum_{i \neq j} \text{sim}(i, j), \tag{1}$$

where $N$ is the number of recommended items, and $\text{sim}(i, j)$ represents the similarity between items $i$ and $j$.

- **Higher ILD**: Indicates more diverse recommendations, suggesting a broader content exposure for users.
- **Lower ILD**: Suggests recommendations are highly similar, potentially limiting content discovery.

To maintain consistency across users, we compute ILD for the top-10 recommended items per user. Item similarity is measured using cosine similarity between post feature embeddings. The final ILD score is obtained by averaging across all users in the validation set.

### 3.2.2 Comparison of Inference Performance

Since our task focuses on "like" recommendation rather than passive "view" recommendation, it inherently presents a greater challenge. Unlike views, which require minimal user effort, likes necessitate additional engagement, making it more difficult to predict user preferences accurately. Consequently, reasonable performance ranges for our evaluation metrics differ from those used in conventional view-based recommendations. Additionally, performance can vary depending on dataset characteristics, model architecture, and evaluation conditions. Thus, offline evaluation should be complemented with online testing to ensure robust real-world performance. However, since we do not have access to an offline evaluation setup, we report the values based on the available test data.

Table 1 presents the inference performance comparison between the three recommendation approaches: Popularity-Based, MLP-Based, and GraphRec. We evaluate each method based on five key metrics: Mean Reciprocal Rank (MRR), Average Rank, Hit@10, ILD@10, Training Time (1 epoch), and Inference Time (seconds per user).

Table 1: Inference performance comparison between different recommendation models. ↑ indicates that a higher value is better, and ↓ indicates that a lower value is better.

| Model | MRR ↑ | Hit@10 ↑ | ILD@10 ↑ | Training (1 epoch) ↓ | Inference (per user) ↓ |
|---|---|---|---|---|---|
| Popularity | 0.0345 | 0.06 | **0.6243** | NA | **0.118 sec** |
| MLP | 0.02 ± 0.001 | 0.03 ± 0.01 | 0.411 ± 0.008 | 10:48 | 0.215 sec |
| GraphRec* | **0.228 ± 0.005** | **0.235 ± 0.01** | 0.55 ± 0.02 | 18:41 | 0.283 sec |

As shown in Table 1, the three recommendation approaches exhibit distinct trade-offs. The Popularity-Based model achieves the best ILD@10 score and the fastest inference time, suggesting that it generates diverse recommendations efficiently. However, its ranking performance is relatively weak, as indicated by lower MRR and Hit@10 values.

The MLP-Based model performs worse than the Popularity-Based model across all three key metrics (MRR, Hit@10, and ILD@10), indicating that it struggles to rank relevant items effectively while also providing less diverse recommendations. Additionally, its training and inference times are higher compared to the Popularity-Based approach, making it a less favorable option.

In contrast, GraphRec achieves the highest MRR and Hit@10, highlighting its effectiveness in ranking relevant items. Although it requires longer training and inference times, it provides a strong balance between ranking accuracy and diversity, as evidenced by its competitive ILD@10 score.

These results underscore the importance of evaluating multiple metrics when selecting a recommendation model, as each approach offers a different balance of accuracy, diversity, and computational efficiency.

# 4 Discussion

In this section, we discuss the trade-offs observed between inference time, ranking accuracy, and recommendation diversity across the three models, and further investigate the relationship between post popularity and the ranking positions.

## 4.1 Trade-off Between Inference Time and Recommendation Quality

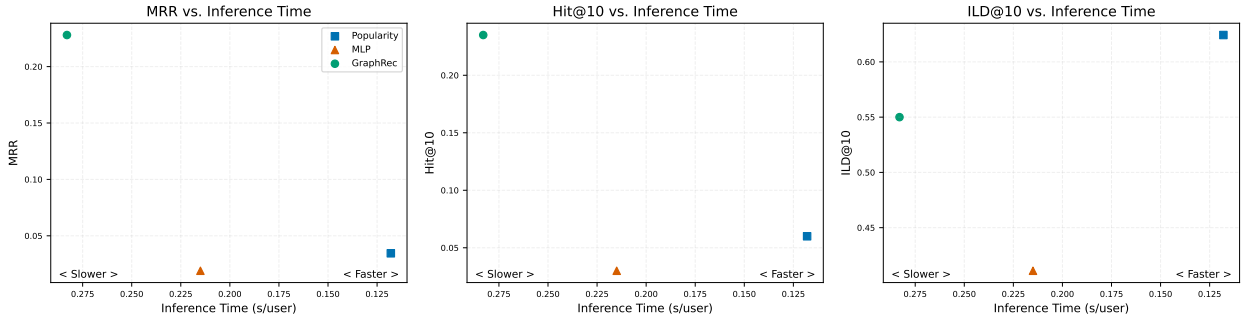



Figure 2: Trade-off Between Inference Time and MRR / Hit@10 / ILD@10

Figure 2 illustrates the trade-off between inference time and three key recommendation performance metrics. This figure provides a visual summary of how each model balances computational efficiency with the quality of recommendations.

The GraphRec model achieves the highest MRR (0.228) and Hit@10 (0.235), demonstrating superior ranking accuracy, while also maintaining competitive ILD@10 scores. An MRR of 0.228 indicates that relevant posts appear, on average, within the top 4 or 5 recommendations, signifying that the model effectively ranks engaging content higher. A Hit@10 value of 0.235 means that for 23.5% of users, at least one of the top 10 recommended posts is relevant, suggesting that GraphRec successfully identifies user-preferred content at a reasonable rate.

However, these benefits come with increased training times and slower inference speeds. Users must wait more than 0.283 seconds to receive recommended posts after refreshing their feeds. This suggests that while GraphRec excels in ranking relevant and personalized content, the associated computational overhead may pose challenges for real-time applications or large-scale systems.

In contrast, the Popularity-Based model achieves the fastest inference time (0.118 sec/user) and the highest ILD@10 score (0.6243). This suggests that the model can generate recommendations rapidly while maintaining a high level of diversity. The high ILD@10 score indicates that users are exposed to a wide range of content, likely because popular posts cover diverse topics. However, this model exhibits a low MRR (0.0345) and Hit@10 (0.06), meaning that while its recommendations are diverse, they are not well-personalized or highly ranked for individual users. The low MRR suggests that when a relevant post is included in the recommendations, it is typically ranked around 30th place on average, requiring users to scroll further to discover engaging content.

The MLP-Based model, on the other hand, struggles to find a balance between accuracy, diversity, and computational efficiency. It achieves a lower MRR (0.019) and Hit@10 (0.03) than both the Popularity-Based and GraphRec models, suggesting that it ranks relevant posts even less effectively. Additionally, its ILD@10 (0.411) is noticeably lower than that of the Popularity-Based model (0.6243) and GraphRec (0.613), indicating that its recommendations are less diverse, which may result in repetitive or redundant content for users. Moreover, the inference time of 0.215 sec/user is almost twice as slow as the Popularity-Based model (0.118 sec/user), yet it does not provide any substantial gains in ranking accuracy or diversity. This suggests that the MLP model fails to leverage complex user–post and user–user relationships effectively, making it a less viable option for balancing recommendation quality and computational efficiency.

In summary, while the Popularity-Based model offers rapid inference and diverse recommendations, it lacks personalization. Meanwhile, the MLP-Based model, despite incorporating post features, fails to achieve competitive accuracy or diversity. These shortcomings highlight the need for more advanced models like GraphRec, which leverages Transformer-based graph architectures to better balance relevance, diversity, and ranking accuracy.

## 4.2 How Do Different Models Rank Popular Posts?

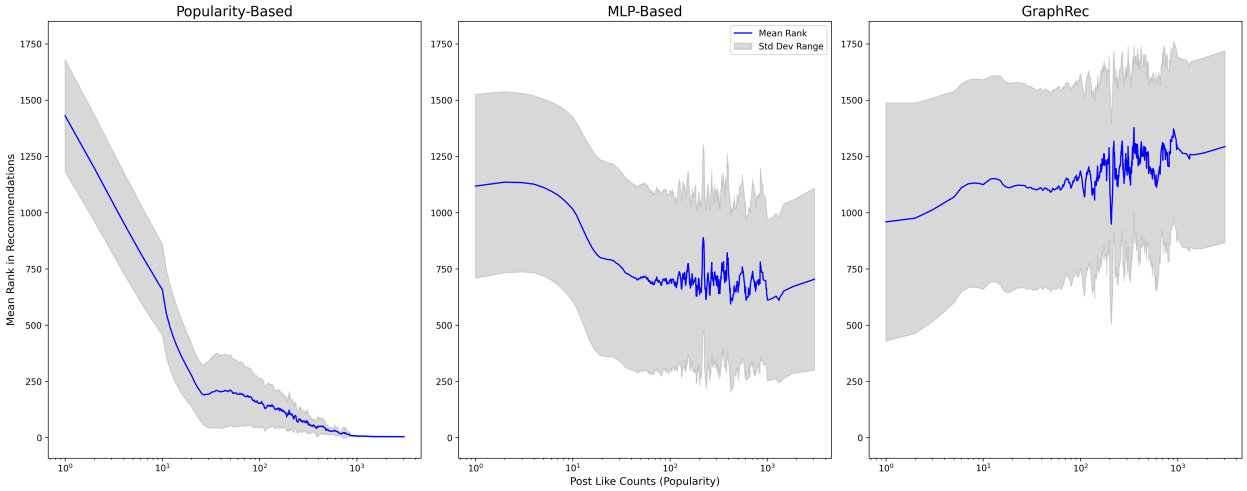



Figure 3: Mean Rank vs. Popularity Across Different Models

To further explore the behavior of our recommendation models, we analyze how post popularity is related to its ranking. Figure 3 shows how the 3 models rank 2,000 candidate posts. The y-axis represents the mean assigned rank across all users, and the x-axis represents the number of likes a post received.

The results indicate that the Popularity-Based model strongly favors popular posts by consistently ranking them at the top. This is an expected behavior since we sort posts by their number of likes. This bias, while effective in promoting content that many users have engaged with, also results in poor visibility for niche or less popular posts. Consequently, users might miss

out on diverse content that aligns with their specific interests, and content creators without a large follower base may struggle to gain visibility and engagement.

The MLP-Based model shows a more balanced approach. By integrating post features and leveraging graph structures, it partially mitigates the popularity bias. While a post without any likes starts at a lower position, once it accumulates approximately 20 to 30 likes, the favoring trend begins to plateau and remains consistent. This indicates that the model makes some effort to reduce the bias toward highly popular posts, allowing lesser-known content to gain visibility once it reaches a moderate level of engagement.

GraphRec, on the other hand, leverages both user–user and user–post interactions to offer a more refined ranking. Unlike the other models, this approach does not inherently prioritize popular posts, allowing it to surface niche content alongside well-liked posts when they align with a user's interests. This results in a more diverse and personalized recommendation list. While this improved balance comes with increased computational cost, the enhanced diversity justifies the trade-off in scenarios where recommendation quality is critical.

Overall, our analysis suggests that while popularity is a strong driver in recommendation rankings, integrating Transformer-based architecture can reduce the bias towards popular content and offer a more balanced set of recommendations. This underscores the importance of considering multiple evaluation metrics when designing and selecting recommendation models, especially when the objective is to ensure both high ranking accuracy and content diversity.

# 5 Conclusion

In this work, we presented a scalable recommendation system tailored for decentralized social networks, using data from Bluesky. Our approach employs a two-stage pipeline, with candidate generation via a simple online collaborative filtering method followed by a ranking phase driven by a graph-based transformer model. Experimental results demonstrate that GraphRec outperforms both the popularity-based and MLP-based methods in terms of ranking accuracy (MRR and Hit@10) and recommendation quality (ILD@10), despite incurring higher computational costs.

The transformer-based architecture in GraphRec effectively captures complex user–post and user–user interactions, leading to more personalized and diverse recommendation lists. These findings highlight the potential of advanced graph transformer models for handling the dynamic and large-scale nature of social media data. However, the increased training and inference times signal a need for further optimization, particularly for real-time applications.

Since we do not have access to offline evaluation and there are no standardized value ranges for MRR, Hit@10, and ILD@10 in like-based recommendation tasks, we cannot definitively determine whether the reported values fall within the expected range of performance. This limitation underscores the importance of future work incorporating online evaluation methods to validate model effectiveness in real-world user interactions.

Future work will focus on reducing computational overhead through improved model architectures and inference techniques, integrating multi-interest embeddings, and incorporating

richer temporal and contextual features.

Overall, our study demonstrates that integrating graph transformer architectures offers a compelling strategy for achieving scalable and personalized recommendations on social media platforms, even as it underscores the need for further optimization to balance computational efficiency with enhanced interaction modeling.

# References


[1] **Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby.** 2021. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale." [Link]

[2] **Nie, Yuqi, Nam H. Nguyen, Phanwadee Sinthong, and Jayant Kalagnanam.** 2023. "A Time Series is Worth 64 Words: Long-term Forecasting with Transformers." [Link]

[3] **Rendle, Steffen, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme.** 2012. "BPR: Bayesian Personalized Ranking from Implicit Feedback." [Link]

[4] **Satuluri, Venu, Yao Wu, Xun Zheng, Yilei Qian, Brian Wichers, Qieyun Dai, Gui Ming Tang, Jerry Jiang, and Jimmy Lin.** 2020. "SimClusters: Community-Based Representations for Heterogeneous Recommendations at Twitter." In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. New York, NY, USA Association for Computing Machinery. [Link]

[5] **Sturua, Saba, Isabelle Mohr, Mohammad Kalim Akram, Michael Günther, Bo Wang, Markus Krimmel, Feng Wang, Georgios Mastrapas, Andreas Koukounas, Andreas Koukounas, Nan Wang, and Han Xiao.** 2024. "jina-embeddings-v3: Multilingual Embeddings With Task LoRA." [Link]

[6] **Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin.** 2023. "Attention Is All You Need." [Link]

[7] **Xu, Da, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan.** 2020. "Inductive representation learning on temporal graphs." In *International Conference on Learning Representations*. [Link]

[8] **da Xu, chuanwei ruan, evren korpeoglu, sushant kumar, and kannan achan.** 2020. "Inductive representation learning on temporal graphs." In *International Conference on Learning Representations (ICLR)*.

[9] **Yu, Le, Leilei Sun, Bowen Du, and Weifeng Lv.** 2023. "Towards Better Dynamic Graph Learning: New Architecture and Unified Library." [Link]