RESEARCH-ARTICLE

Bridging GNN Inference and Dataflow Stream Processing: Challenges and Opportunities

NAIMA ABRAR SHAMI, Boston University, Boston, MA, United States

VASILIKI KALAVRI, Boston University, Boston, MA, United States

Open Access Support provided by:

Boston University

Published: 22 June 2025

SIGMOD/PODS '25: International Conference on Management of Data
June 22 - 27, 2025
Berlin, Germany

Conference Sponsors:
SIGMOD

GRADES-NDA '25: Proceedings of the 8th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (June 2025)
https://doi.org/10.1145/3735546.3735856
ISBN: 9798400719233

Bridging GNN Inference and Dataflow Stream Processing: Challenges and Opportunities

Naima Abrar Shami
Boston University
anaima@bu.edu

Vasiliki Kalavri
Boston University
vkalavri@bu.edu

Abstract

Graph Neural Networks (GNNs) have the potential to address real-world problems involving continuously evolving, graph-structured data, such as fraud detection, real-time recommendations, and traffic monitoring. These applications require the timely processing of streaming (possibly unbounded) data, highlighting the need of integrating GNN inference with dataflow stream processing systems, like Apache Flink. In this paper, we present the first exploration of bridging this gap, by designing a streaming GNN serving pipeline with Flink and PyTorch. We propose a dataflow architecture that offloads subgraph construction to Flink, leveraging its state management and distributed processing capabilities. Despite achieving viable performance through asynchronous inference requests and careful parallelism tuning, we also identify significant limitations stemming from Flink's state scoping, lack of iterative processing, and computation pipelining. We propose solutions that mitigate these issues within Flink and discuss open challenges towards developing dataflow systems tailored to streaming GNN inference.

CCS Concepts

• **Information systems** → **Stream management**; • **Computing methodologies** → **Machine learning**.

Keywords

Graph Neural Networks, Stream Processing

ACM Reference Format:
Naima Abrar Shami and Vasiliki Kalavri. 2025. Bridging GNN Inference and Dataflow Stream Processing: Challenges and Opportunities. In *8th Joint Workshop on Graph Data Management Experiences Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3735546.3735856

1 Introduction

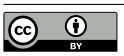
Graph Neural Networks (GNNs) have emerged as a powerful paradigm for processing and analyzing graph-structured data. They are widely used in domains such as drug discovery [21], recommendation systems [23, 37, 38], and traffic optimization [11, 19]. Unlike traditional machine learning models that operate on tabular or sequential data, GNNs capture the rich relational structure of graphs by aggregating information from neighboring nodes through iterative message passing. This enables them to learn expressive representations for nodes, edges, or subgraphs; supporting tasks like node classification, link prediction, and personalized ranking.

While GNNs have shown strong performance in offline settings, many real-world applications require inference in real time, as new nodes and edges arrive. Consider fraud detection in financial systems [20]: banks and payment platforms process millions of transactions daily, forming a streaming transaction graph, where nodes represent accounts and edges represent money transfers. As new transactions arrive, the system must quickly retrieve the relevant subgraph (e.g. involving the sender and receiver), update embeddings, and assess the transaction for suspicious patterns – all within milliseconds before the transaction is finalized. Similarly, in e-commerce and content platforms [32], real-time GNN inference enables updating user and item embeddings on-the-fly, to serve personalized recommendations that reflect current behavior rather than stale history. These workloads require fast, localized subgraph construction and inference in response to streaming data [27].

Dataflow stream processing systems [25, 29, 30], like Apache Flink and Spark Streaming, support continuous ingestion, low-latency data processing, event-time semantics, and window-based operations over unbounded streams. At the same time, they provide efficient distributed execution, enabling the development of large-scale GNN inference pipelines, whose load may exceed the capacity of a single machine. However, current dataflow systems do not natively support GNN operations, such as k-hop neighborhood traversal, subgraph assembly, or integration with deep learning frameworks [15]. Bridging this gap is challenging because GNNs are inherently recursive, while dataflow systems typically support acyclic computation graphs.

In this paper, we explore the systems challenges of performing GNN inference (using a pre-trained model) within a streaming dataflow framework. As a first step, we design and implement a prototype pipeline that operates on a static graph snapshot, preloaded into Flink's managed state. The pipeline offloads subgraph construction to Flink and performs inference using a pre-trained GraphSAGE model hosted in TorchServe. Inference requests arrive to the system as a stream of target node IDs, which trigger node-level prediction tasks. This setting is representative of applications that aim to classify or score individual entities (e.g. users or accounts), such as fraud detection or recommendation. For each such node, Flink (i) retrieves its k-hop neighborhood and features from distributed keyed state and (ii) sends the resulting subgraph to TorchServe for embedding generation and prediction.

In developing this streaming GNN pipeline, we identify several fundamental challenges. First, GNN inference requires iterative neighborhood expansion, but Flink does not support iteration natively, hence, each hop must be represented by an explicit operator stage. Second, storing the graph data in keyed state [8] restricts



GRADES-NDA '25, Berlin, Germany

ACM ISBN 979-8-4007-1923-3/2025/06
https://doi.org/10.1145/3735546.3735856

operators from directly reading the state of neighboring nodes and forces additional communication and repartitioning. Third, due to pipelined and parallel execution, operators lack a built-in notion of when a subgraph is *complete*, complicating neighborhood assembly before inference. Finally, because Flink relies on embedded state backends, like RocksDB, the graph state must be initialized by a separate job, which increases the pipeline complexity.

We evaluate our prototype on a real-world graph and demonstrate that it can sustain over 2000 requests/s, with median end-to-end latency under 400ms, across a range of configurations. We test how system throughput and latency vary under different input rates, hop limits, and model configurations, and highlight open research directions for extending dataflow systems to support graph-centric and recursive workloads required by GNN inference.

We make the following contributions:

- We design the first Flink-based GNN inference pipeline that performs subgraph construction in streaming mode and integrates with TorchServe for real-time embedding generation.
- We identify and categorize fundamental challenges in using dataflow systems for GNN inference.
- We propose simple yet effective mechanisms to overcome these challenges using Flink's existing abstractions.
- We discuss open research problems and opportunities for future work towards making streaming GNN scalable.

Our code is available at https://github.com/CASP-Systems-BU/GNN_InferenceStream.

2 Background

In this section, we briefly review the GNN inference process, introduce key concepts of stream processing systems, and explain how GNN inference can be mapped to the streaming dataflow model.

2.1 Overview of GNN inference

GNNs are a class of neural networks that operate on graph-structured data [14, 33, 42]. The core principle behind GNNs is the iterative aggregation of features from a node's neighborhood, enabling nodes to learn representations called *embeddings*, which capture local and global properties of the graph. GNN inference is the process of generating these embeddings for nodes (or edges or entire graphs) using a pre-trained GNN model. In this work, we focus specifically on sampling-based GNNs [9, 14, 37], where a subset of a node's neighborhood is sampled for aggregation during inference. We use GraphSAGE [14] as a representative sampling-based GNN model.

The inference process typically consists of three stages, as shown in Figure 1. First, for a given *target* node (the node for which we aim to compute an embedding), its k-hop neighborhood is retrieved from the graph. This neighborhood includes both the node IDs and their associated feature vectors. Second, features from neighboring nodes are aggregated layer by layer, where each layer corresponds to a hop in the neighborhood. This stage, known as message passing and aggregation, is often implemented as a sequence of matrix multiplications and non-linear activation functions in frameworks such as PyTorch Geometric [9] or DGL [3]. Finally, after k layers of aggregation, the model computes the embedding of the target node.

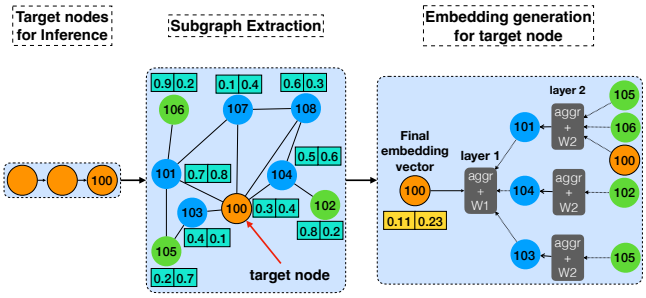


Figure 1: GNN inference process. The figure illustrates the three stages of inference: streaming of target nodes, extracting k-hop neighborhood, and embedding generation.

This embedding can then be used for downstream tasks, such as node classification or link prediction.

While traditional GNN inference is often performed in batch settings over static graphs [28, 35, 40], many real-world applications require performing inference *online*, on a stream of target nodes (or edges). In such scenarios, retrieving neighborhoods and generating embeddings must be performed continuously, as new data arrives. This necessitates a system capable of continuous ingestion, stateful processing, and scalable execution – characteristics provided by dataflow stream processing frameworks, like Apache Flink.

2.2 The dataflow stream processing model

Stream processing systems (SPSs) [13, 29, 30] are designed to process unbounded, continuously arriving data with low latency and high throughput. In the context of online GNN inference, data typically arrives as a stream (e.g. user events, financial transactions) and might require stateful pre-processing, complex feature engineering (potentially derived from multiple streams or joins with graph data), or filtering, before triggering predictions. SPSs excel at building and managing such integrated pipelines.

In this work, we select Apache Flink, as a representative dataflow stream processing system. A *dataflow program* in Flink is a directed acyclic graph (DAG) where vertices represent operators and edges represent the flow of data between them. At the logical level, a Flink job is defined as a sequence of transformations, such as filtering, projection (extracting specific fields), grouping (partitioning data based on a key), continuous aggregation (maintaining running computations), joins (combining multiple streams), and windowed aggregation (aggregating data within time intervals). These logical operations are then translated into a physical dataflow and executed in parallel, across a distributed cluster.

An important concept in the dataflow execution model is *keyed state* [8]. Dataflow systems partition incoming records based on a key (e.g. a node ID in a graph) and assign physical partitions to parallel operator tasks. The system runtime transparently routes records with a specific key to its corresponding physical partition. The state associated with each key is isolated and stored locally within each task. To enable persistence and scalability, Flink uses an embedded key-value store, such as RocksDB, to manage large state efficiently. This design enables operators to maintain large amounts of state without requiring all data to reside in memory.

Modern dataflow systems also support *event-time* processing, where the computation progress in the system is determined by *watermarks* [2]. Watermarks allow dataflow tasks to reason about time-dependent operations, such as timers and windows, in the presence of out-of-order events. Operators can register *event-time timers* that trigger actions, like window aggregation or other computations, once the watermark surpasses a predefined timestamp.

2.3 GNN inference as a streaming dataflow

At a high level, GNN inference can be conceptually mapped to streaming dataflow program. The three primary stages of GNN inference – (i) neighborhood construction, (ii) feature retrieval, and (iii) embedding generation—naturally correspond to SPS transformations. However, mapping GNN inference to a dataflow computation requires representing the graph structure (nodes and edges) and node features as key-value pairs. In such a representation, each node ID serves as a key, while the corresponding value stores the node's feature vector and its adjacency information. When a SPS receives a request to generate an embedding for a target node, it must execute a series of operations that mirror the GNN inference process: retrieve the k-hop neighborhood around the target node, collect the feature vectors for all nodes in this neighborhood, and finally compute the embedding by applying the learned GNN parameters to this localized subgraph. Realizing this conceptual mapping presents several practical challenges, particularly in managing distributed state for large graphs, efficiently expanding k-hop neighborhoods across parallel operators, and maintaining low-latency, high-throughput processing in a streaming environment. We explore these challenges in detail in the following section.

3 Challenges of streaming GNN inference

Before diving into the technical challenges of implementing GNN streaming inference, we first describe how the graph is represented and stored as dataflow state. Systems like Flink require that all state be explicitly partitioned by key, so we represent the graph with two key-value mappings: (1) a structure mapping that captures the graph topology as `nodeID` $\rightarrow$ `list of neighbors`, and (2) a feature mapping that stores node attributes as `nodeID` $\rightarrow$ `feature vector`. This representation allows the graph to be distributed across multiple parallel workers based on node IDs and enables local neighborhood lookups during subgraph construction. Table 1 illustrates how the example graph of Figure 1 would be represented internally as key-value pairs in Flink state.

Table 1: Key-value representation of the graph in Figure 1.

(a) Structure mapping

Key (Node ID)	Value (Neighbors)
100	[101, 103, 104, 107, 108]
101	[100, 105, 106, 107]
102	[104]
103	[100, 105]
104	[100, 102, 108]
105	[101, 103]
106	[101]
107	[100, 101, 108]
108	[100, 104, 107]

(b) Feature mapping

Key (Node ID)	Value (Feature Vector)
100	[0.3, 0.4]
101	[0.7, 0.8]
102	[0.8, 0.2]
103	[0.4, 0.1]
104	[0.5, 0.6]
105	[0.2, 0.7]
106	[0.9, 0.2]
107	[0.1, 0.4]
108	[0.6, 0.3]

3.1 Challenge 1: Lack of iterative support

Stream processing frameworks rely on a static dataflow architecture, where the computational pipeline is pre-defined as a directed acyclic graph (DAG) of operators. However, graph traversal for GNN inference requires retrieving a k-hop neighborhood, repeatedly exploring neighbors around a node for k steps. This iterative graph traversal is in conflict with the acyclic structure of the dataflow model. As a result, each hop in the traversal must be explicitly represented as a separate operator stage. For instance, if the pipeline is designed to expand up to 3 hops, it must explicitly define three distinct operators in the job. This design constraint prevents dynamically adjusting the traversal depth without redefining and redeploying the pipeline, complicating support for evolving inference requirements for different applications.

3.2 Challenge 2: Keyed state prevents k-hop neighborhood access

Flink manages distributed state by partitioning data into distinct keyed partitions. Operators process events under an active key, and state access is strictly scoped to that key [8, 17]. Consequently, an operator handling a given node can only directly access its own associated state (1st-hop neighbors and features). For GNN inference, retrieving k-hop neighbors requires accessing the states associated with multiple nodes. However, because of keyed isolation, operators cannot directly read the state of neighboring nodes. Instead, the operator must emit additional requests to downstream operators explicitly keyed by these neighbors. This limitation introduces extra steps: each traversal step involves signaling downstream operators to perform state lookups for specific keys (neighbor IDs), incurring overhead due to additional message passing, and network shuffles. While the keyed state design facilitates parallel processing and scalability, it fundamentally limits expressiveness. Operators cannot directly retrieve the state of multiple keys in a single context, making it impossible to perform k-hop neighborhood aggregation as a local operation.

3.3 Challenge 3: Subgraph completion

Streaming dataflow operators do not have advance knowledge of the full set of events associated with a computation. This creates a fundamental challenge for GNN inference: there is no built-in signal to indicate when all the nodes and edges of a k-hop neighborhood have arrived. Unlike batch systems where subgraphs can be precomputed, streaming systems must assemble them incrementally from continuously arriving events. Moreover, shuffle operations (e.g. `keyBy`) introduce data re-partitioning across parallel downstream operators, which combined with pipelined and distributed execution can result in messages arriving out of order. This lack of coordination and out-of-order delivery complicates subgraph assembly and makes it difficult to detect when a neighborhood is complete. These challenges arise from two factors:

(1) Pipeline parallelism: Subgraph components such as neighbors at different hops are processed concurrently by multiple parallel operator instances. There is no inherent coordination to signal when all necessary parts of a specific subgraph have been collected.

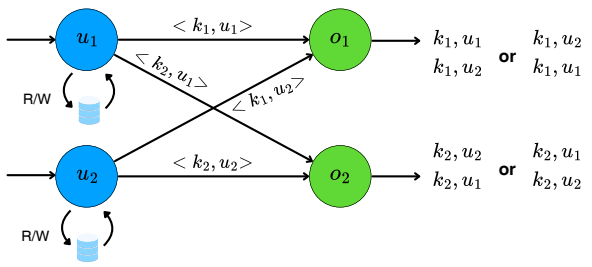


Figure 2: Data parallelism and out-of-order delivery in streaming systems

(2) Data parallelism and out-of-order delivery: Distributed execution introduces arbitrary delays in data propagation. As a result, nodes from deeper hops (e.g., second-hop neighbors) may arrive before immediate neighbors, disrupting the dependency order required for accurate GNN feature aggregation. Figure 2 illustrates this challenge. Two upstream tasks, processing nodes u_1 and u_2, independently emit messages containing neighbor information (tuples $\langle k, u \rangle$) destined for downstream tasks o_1 and o_2. Each downstream task is responsible for a distinct key: o_1 aggregates messages for key k_1, while o_2 aggregates messages for k_2. However, due to shuffle operations and data parallelism, the messages may arrive in arbitrary order. For instance, o_1 may receive u_1 before u_2, or vice versa. In addition, the downstream operators do not know when they have received all messages related to their respective keys. That is, there is no explicit signal indicating subgraph completeness. This uncertainty complicates GNN inference, as incomplete or prematurely processed subgraphs may result in missing neighborhood information during aggregation.

3.4 Challenge 4: Graph state initialization

While any data-intensive application must perform a data loading and preparation phase, if it depends on existing data, graph state loading requires special care in Flink, as we explain below. GNN inference requires access to both the model weights and the underlying graph structure that the model was trained on. While model weights can be loaded independently by the inference server (e.g. TorchServe), the graph structure and node features must be available to Flink operators at run time. Because Flink manages application state via embedded RocksDB instances, this data must first be ingested into Flink's keyed state, where each node is stored as a key-value pair; with the node ID as the key and its feature vector or adjacency list as the value. While it is technically possible to query external databases for node features or edges at inference time, doing so introduces additional latency and is not compatible with Flink's state processing guarantees. Executing the required initialization job involves significant framework-specific complexity, beyond standard data loading. The job must ingest the graph data and populate Flink's keyed state using specific APIs (e.g. `ListState` and `ValueState`), while it must also ensure correct state restoration by the subsequent inference job.

4 Solution Overview

To address the challenges of real-time GNN inference described in Section 3, we design a distributed pipeline using Apache Flink and TorchServe. Our architecture performs 2-hop subgraph extraction for incoming nodes, constructs feature-rich subgraphs in JSON format, and sends them to a GraphSAGE model for inference. The pipeline consists of two Flink jobs: a *DataLoader* job that initializes Flink's graph state and an *Inference* job, shown in Figure 3, that handles streaming inference.

Pipeline overview. The system executes the following steps:

- The **DataLoader job** ingests the training graph and partitions it across parallel subtasks. Node features and adjacency lists are stored in Flink's keyed state, backed by RocksDB (cf. **Challenge 4**).
- The **Inference job** restores this state from a savepoint [5] and consumes a stream of target node IDs, generated by a custom source simulating real-time inference requests.
- For each target node, the job fetches 1-hop and 2-hop neighbors and features, by successively re-keying and accessing state in separate operator stages (cf. **Challenge 1, 2**).
- To address the absence of explicit subgraph completion signals in Flink (cf. **Challenge 3**), the system uses event-time watermarks and timers to detect when all relevant neighbor data has arrived.
- Once complete, the neighborhood is assembled into a JSON subgraph containing node features and edge lists. This subgraph is sent to a GraphSAGE model hosted on TorchServe for inference.
- The resulting embedding is published to a Kafka sink topic for downstream use.

Next, we describe each component of the pipeline in detail.

Graph state initialization. The *DataLoader* job ingests training graph data from a file or Kafka source, where each record includes a node ID, a list of its immediate neighbors, and its feature vector. These records are partitioned using Flink's `keyBy` operator, and the graph is stored in RocksDB keyed state: neighbors are saved in `ListState` and features in `ValueState`, both keyed by node ID. This design allows the graph to be partitioned and distributed across parallel subtasks. However, as discussed in Challenge 3, Flink does not support shared state or direct reads across keys or operators. As a result, we load the graph state with a dedicated job and materialize it as embedded keyed state, ensuring that the subsequent Inference job can restore this state consistently via a savepoint.

Target node ingestion. The *Inference* job restores its state from the savepoint and receives a stream of target node IDs through a custom source. This generator emits target nodes at a configurable rate, where each record includes a node ID and a timestamp to simulate real-time event arrival.

Neighborhood retrieval. To construct a subgraph around the target node, the Inference job retrieves both its 1-hop and 2-hop neighbors using two chained operators. Since Flink does not support iterative graph traversal or cyclic dataflows, we explicitly model each hop as a separate stage in the dataflow graph. The first operator retrieves the immediate neighbors from `ListState`, emitting them

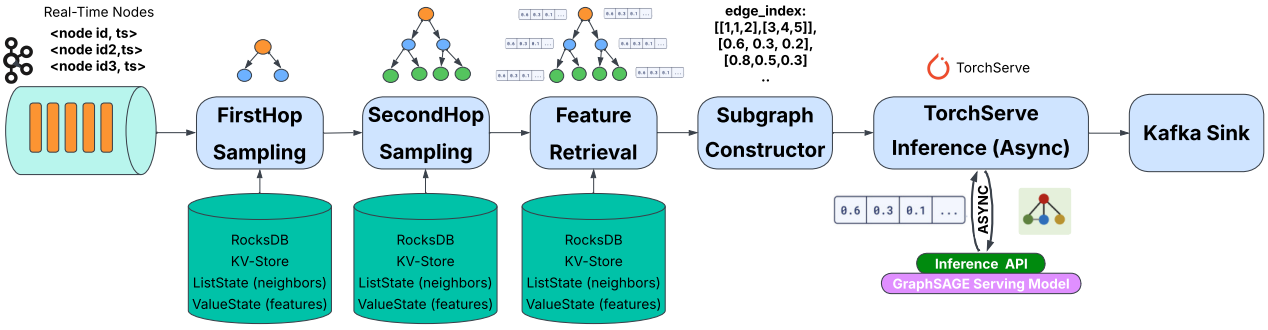


Figure 3: End-to-end pipeline for streaming GNN inference. The DataLoader job initializes graph state in Flink, while the Inference job processes streamed node IDs to retrieve neighborhoods and perform inference via TorchServe.

as first-hop nodes. These are re-keyed and sent to a second operator that fetches the neighbors of the first-hop nodes—thereby retrieving second-hop neighbors. This unrolling of hops avoids the need for loop constructs, working around Flink's acyclic execution model (Challenge 1). Because Flink enforces strict isolation of keyed state, each operator stage must access only the state associated with its active key. To retrieve neighbors at each hop, we emit neighbor IDs downstream and re-key the stream accordingly. Each stage must independently access state, and since Flink requires each stateful operator to be uniquely identified by a *UID* (a user-assigned identifier that binds the operator to its state), we duplicate the neighbor and feature state for each stage during initialization. This ensures that each operator in the streaming job can correctly restore its portion of state from the DataLoader savepoint.

Feature retrieval. As neighbor IDs are collected from the first and second-hop sampling stages, they are immediately passed downstream to a keyed operator for feature retrieval. Each node ID—whether the target or a 1/2-hop neighbor is looked up in `ValueState` to retrieve its feature vector. The result is a stream of tuples, each containing the target node ID, the source node that emitted the current node (used for tracking neighborhood connectivity), the current node ID whose features are being retrieved, the feature vector, and the hop level (1 for 1-hop neighbor, 2 for 2-hop).

Subgraph assembly. To determine when all parts of a subgraph have arrived at a subtask, we rely on Flink's event-time processing model [6]. Each target node is assigned an event-time timestamp, and a timer is registered to fire after the watermark passes this time plus an allowed lateness (eg, +1ms). This mechanism ensures that all relevant messages for the subgraph have been received before proceeding with assembly. The subgraph is assembled in a `KeyedProcessFunction` that collects all relevant records into a buffer. When the timer fires, we serialize the subgraph into a JSON object containing node features and edge indices. TorchServe expects input in a structured format, and JSON provides a flexible, language-agnostic way to package the subgraph for inference.

GNN inference and output. The serialized subgraphs are sent to TorchServe with Flink's asynchronous API [4]. The model returns the embedding for the target node, which is then published to a Kafka sink topic for downstream consumers, such as a recommendation engine or fraud detection module.

This design carefully adapts Flink's dataflow model to support recursive, stateful operations required by GNN inference. By leveraging Flink's keyed state, savepoints, and event-time processing, we demonstrate that subgraph-based GNN inference is feasible within a stream processing system, despite the abstraction gap between graph-based computation and acyclic dataflows.

5 Experimental evaluation

We evaluate our prototype to understand how key system parameters impact the performance of streaming GNN inference. Several factors may influence system behavior: (i) TorchServe worker concurrency determines how many inference requests can be served in parallel; (ii) the neighborhood size (i.e., the number of hops and fanout at each hop) affects subgraph size, which in turn impacts state access, data movement, and inference time; (iii) the input load (inference requests per second) stresses the system's ability to keep up with high arrival rates; and (iv) the inference mode (synchronous vs asynchronous) affects throughput by determining whether HTTP requests block or are executed concurrently. TorchServe also supports request batching through two parameters: `batch_size` and `max_batch_delay`. The `batch_size` controls how many subgraphs are grouped into a single inference request, while `max_batch_delay` sets the maximum wait time (in milliseconds) to collect a batch before it is processed. In all experiments, we set `batch_size = 1` and `max_batch_delay = 100ms`. Empirically, we observed that increasing these values did not significantly improve throughput or latency in our pipeline. Hence, we focus our evaluation on the three parameters that had the most significant effect: TorchServe worker concurrency, subgraph size (controlled by hop limits), and input rate. Our experiments aim to answer the following questions:

- How does TorchServe concurrency affect maximum sustainable throughput?
- How does varying neighborhood size affect throughput and latency?
- What is the distribution of end-to-end latency under sustained load?

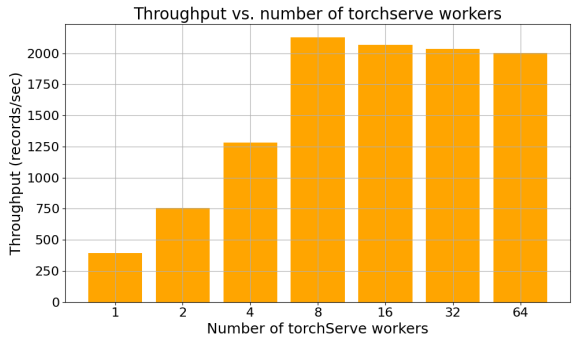


Figure 4: Throughput (records/second) as the number of TorchServe model workers increases

- How does asynchronous inference compare to synchronous inference in terms of throughput?

We record the throughput using Flink's `numRecordsOutPerSecond` metric, and compute latency CDFs by measuring the time from when a node ID is ingested to when its embedding is produced.

5.1 Experimental setup

We conduct our experiments on a server equipped with a 64-core AMD EPYC 7713 CPU (1 thread per core), 251 GB of RAM, and an NVIDIA A100 PCIe GPU with 80 GB of memory. The system runs Ubuntu 20.04.6 LTS with Linux kernel 5.4.0-212-generic, uses Python 3.8.10 for all Python-based components, CUDA 12.2, and PyTorch 2.1.0. For stream processing, we use Apache Flink version 1.20.0 in standalone mode. The pre-trained GraphSAGE model is served using TorchServe within a Docker container with GPU acceleration enabled. To eliminate network overhead, both Flink and TorchServe are deployed on the same machine.

The input graph is the Reddit dataset [7], containing 232,965 nodes and 114,615,892 edges, with each node represented by a 602-dimensional feature vector. To simulate streaming input, we implement a custom source function in Flink that continuously emits node ID events. Each event is a comma-separated string containing a node ID and a timestamp. The node IDs are sampled uniformly at random from the valid range $[0, 232964]$, matching the ID space of the Reddit dataset. The source emits a fixed number of events every 100ms, allowing us to control the input rate by adjusting the number of events per burst. For example, emitting 250 events per burst results in a source rate of 2500 node IDs/second. This setup allows us to evaluate system behavior under different load conditions and node distributions. Flink is configured with 80 GB of task manager memory and 64 task slots, using RocksDB as the state backend. We also set the parallelism for the subgraph constructor to 64. Watermarks are assigned with a maximum out-of-orderness of 100 ms. Checkpointing is disabled to reduce measurement overhead. Unless otherwise specified, we use a 2-layer GraphSAGE model with a (10, 5) hop configuration.

5.2 Effect of parallel TorchServe workers

This experiment evaluates how the number of TorchServe model workers affects the maximum sustainable throughput of our GNN inference pipeline. We incrementally increase the node ID input rate until the pipeline shows signs of backpressure in Flink. Backpressure is Flink's built-in flow control mechanism: if an operator cannot keep up with its incoming data rate, it signals its upstream sources to slow down. This prevents system overload (e.g. running out of buffers) but indicates that the backpressured operator has become a bottleneck, thus limiting the maximum throughput the entire pipeline can sustain. We record the highest input rate (in records/second) that the system can sustain without triggering backpressure. We refer to this value as the *maximum sustainable throughput*. TorchServe uses model workers as independent Python processes, each capable of handling model inference requests concurrently. Since each worker can process a subgraph independently, we expect that increasing the number of workers should improve throughput, up to the point where another bottleneck (e.g. GPU contention or I/O) is reached.

We vary the number of TorchServe workers across {1, 2, 4, 8, 16, 32, 64}. We run each configuration for 5 minutes and exclude the first and last minute from metrics to account for warm-up and shutdown artifacts. Throughput is measured using Flink's internal metric `numRecordsOutPerSecond`, collected at 5-second intervals.

As shown in Figure 4, the throughput increases with the number of workers, reaching a peak at 8 workers with over 2100 records/sec. Beyond 8 workers, the throughput plateaus and even slightly declines, indicating diminishing returns from additional concurrency. With a single worker, the system handles under 400 records/sec, whereas 8 workers deliver more than a 5× improvement.

We believe that the throughput plateau around 8 workers suggests that the bottleneck shifted from TorchServe inference to another other component, such as subgraph construction on the Flink side. While scaling up to 16 or 32 workers did not result in significant performance degradation, it also did not yield additional throughput, despite higher resource consumption. This suggests diminishing returns beyond the 8-worker configuration. These results highlight the importance of server-side parallelism in model inference. While TorchServe concurrency significantly improves throughput, it does not scale linearly, and tuning must account for diminishing returns beyond a certain point.

In Figure 5, we also plot the end-to-end latency distribution for 4, 8, 16, 32, and 64 workers. The results show that latency drops sharply between 4 and 8 workers, with median latency dropping from over 1800ms to under 600ms. As the number of workers increases further, median latency stabilizes, but variance increases again at 32 and 64 workers.

5.3 Impact of hop count on throughput

This experiment evaluates how the size of the neighborhood, which is controlled by the hop sampling parameters, affects our prototype's throughput. In GNN inference, larger neighborhoods result in more data transfer, state access, and computation per request, which can directly impact overall performance. Based on the optimal configuration found in Section 5.2, we fixed the TorchServe worker concurrency at 8 for these tests. We increase the source rate to 3000 node IDs/sec. This is higher than the 2500/sec used earlier, as smaller neighborhoods (e.g. 5-5) result in lower per-request overhead and can sustain higher input rates. We run each

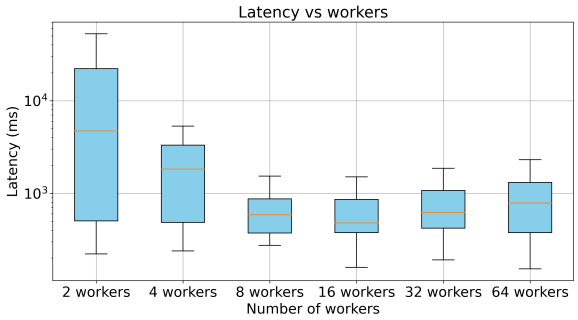


Figure 5: End-to-end latency distribution for different TorchServe worker counts

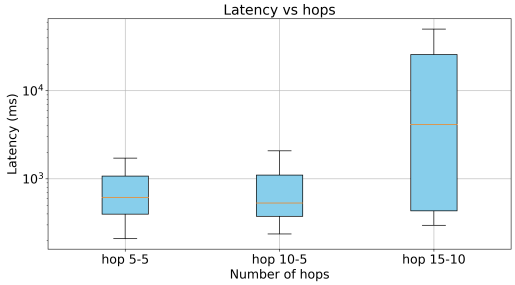


Figure 6: Effect of hop counts on latency

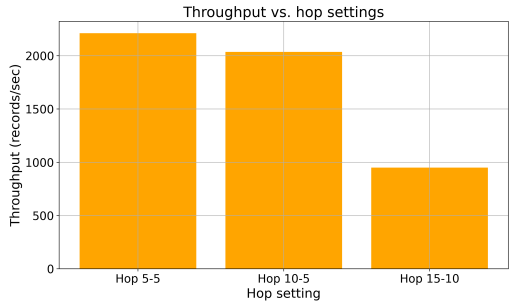


Figure 7: Effect of hop counts on throughput

configuration for 5 minutes and collect Flink throughput metrics (`numRecordsOutPerSecond`) every 5 seconds. After discarding the first and last minute to remove warm-up and tail effects, we report the average sustained throughput for each configuration. Figure 7 shows that throughput decreases significantly as the neighborhood size grows. With the smallest configuration (5-5), the system achieves a throughput above 2200 records/sec. Increasing to (10-5) leads to a moderate drop to just about 2100 records/sec. With the largest neighborhood (15-10), the throughput drops sharply to under 1000 records/sec. These results highlight the sensitivity of the streaming GNN inference pipeline to subgraph size.

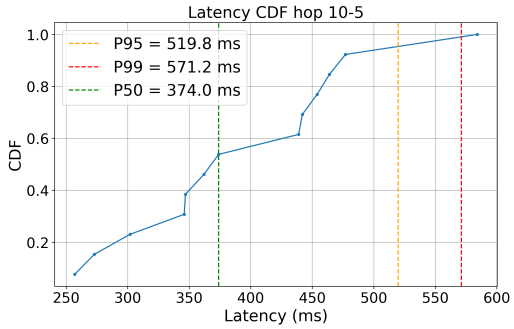


Figure 8: CDF of end-to-end latency

We also examine how neighborhood size impacts latency by analyzing the distribution of end-to-end latencies under each configuration. Figure 6 shows the latency boxplots for three hop configurations: (5, 5), (10, 5), and (15, 10). Smaller neighborhoods lead to faster inference: the (5, 5) configuration has a lower median latency and smaller tail compared to (10, 5) and (15,10). With (5, 5), most requests complete in under 1s. The (10, 5) configuration shows a slightly higher median and more variability, while (15, 10) introduces a wide spread and long tail. This behavior is consistent with our throughput results. As the number of sampled neighbors grows, Flink must perform more state lookups and emit more messages across operators, which increases load on both the Flink runtime and the inference server. Moreover, larger subgraphs result in larger JSON payloads and longer inference times on TorchServe.

5.4 End-to-end latency at sustained load

The goal of this experiment is to assess the responsiveness of the system from the user's perspective and estimate how long a user waits for a prediction after submitting a request. Specifically, we aim to evaluate the distribution of per-request latencies under a specific configuration. We fix the configuration to a (10, 5) fanout and set the source rate to 2100 node IDs per second. This rate is chosen based on our earlier throughput experiment, where we found that 8 TorchServe workers can sustain this load for the (10, 5) setting, without inducing backpressure.

We measure the end-to-end latency for each request from the time a node ID is ingested by Flink to the time the corresponding embedding is returned by the model. Latencies are sampled every 5s over a 5-minute run, with the first and last minute discarded. Figure 8 shows the CDF of end-to-end latencies, annotated with the P50, P95, and P99 latency percentiles. We observe that 50% of the requests complete in under 374 ms, while 95% complete within 519 ms, and 99% within 571 ms. These results indicate that the system offers responsive inference under steady load.

5.5 Effect of asynchronous inference

In our final experiment, we compare synchronous and asynchronous inference strategies to evaluate their effect on throughput. In the synchronous strategy, each subgraph is sent to the model server using a blocking HTTP request. The system waits for a response before proceeding to the next subgraph. This introduces

idle time, as no other work can be performed while waiting for the model's output. In contrast, the asynchronous strategy issues non-blocking HTTP requests to the model server. Instead of waiting, it immediately continues processing other subgraphs while previous requests are still in flight. Once responses arrive, the corresponding results are collected and integrated into the pipeline. This design enables concurrent inference requests, allowing the system to overlap communication with computation. We found asynchronous inference significantly improves system throughput, reaching over 2100 records/s compared to 1550 records/s with the synchronous version.

This improvement stems from the ability to overlap HTTP I/O with other computation, mitigating the blocking behavior of synchronous HTTP requests. These results suggest that incorporating asynchronous execution into the inference pipeline is a key optimization for achieving higher throughput.

6 Related work

Several recent systems aim to improve the efficiency of GNN inference, particularly for large graphs [24] and latency-sensitive applications. Most of this work falls into three categories: (1) systems that optimize inference over static graphs, (2) frameworks that support real-time or incremental inference on dynamic graphs, and (3) model-level optimizations that reduce the computational footprint of inference. Below, we summarize key directions in each space and contrast them with our stream processing approach.

A central bottleneck in GNN inference is the neighbor explosion problem, where node-wise inference triggers exponential neighborhood retrieval. Systems like Deep Graph Inference (DGI) [36] address this by switching from node-wise to layer-wise inference, enabling intermediate reuse and efficient batching across large graphs. These designs often achieve speedups orders of magnitude faster than naive methods but assume that the graph is static and stored in memory or a central store. They are ill-suited to settings where the graph is dynamic, such as in streaming workloads. To support real-time applications like fraud detection or recommendations, recent work has explored incremental inference on streaming graphs. For instance, InkStream [31] avoids full recomputation by updating only the embeddings of nodes impacted by incoming graph mutations. While this approach achieves high throughput, it relies on monotonic aggregators (e.g. min, max), which limits its applicability to models like GraphSAGE or GAT that use mean, attention, or learned aggregation. λGrapher [16] and Quiver [28] target infrastructure-level optimizations. λGrapher focuses on request-level efficiency by caching subgraphs and decoupling aggregation and update stages using serverless primitives. Quiver adapts graph sampling and feature aggregation across CPUs and GPUs based on workload statistics. More broadly, integrating various ML models efficiently into stream processing systems requires addressing general performance and integration challenges, benchmarked by frameworks like Crayfish [15] and explored in specialized serving systems using streaming concepts [39].

Aside from system level improvements, a complementary line of work targets the GNN model itself. Input feature pruning [34] and channel pruning [41] reduce inference cost by removing redundant input features or intermediate channels, often using LASSO-based methods. These approaches are architecture-agnostic and applicable to both training and inference, but they do not solve the system-level challenge of building subgraphs or integrating with event driven pipelines.

In contrast, we present a first exploration of GNN inference within a streaming dataflow system (Apache Flink), tackling the systems integration challenges involved. Our approach performs inference on-the-fly for ccontinuous streams of target node requests, using Flink's state (currently pre-loaded with a static graph) and operators. This explores an alternative to batch-based methods for applications like fraud detection and recommendation systems. Handling dynamic graph updates and demonstrating distributed scalability are key areas for future development.

7 Discussion and future work

Our work presents a practical blueprint for real-time GNN inference using Apache Flink and TorchServe. While we demonstrate the feasibility of streaming subgraph construction and integration with external models, several open challenges and directions remain.

Currently, our pipeline operates on a static snapshot of the graph. The DataLoader job ingests this graph once at initialization and stores it in Flink's embedded RocksDB backend. This fixed view of the graph means that inference requests regardless of when they arrive, are always served using the same underlying graph state. In real-world applications, however, graphs evolve continuously. New users join the platform, new edges form, and new interactions occur. A key direction for future work is supporting dynamic graph updates: enabling edges, nodes, and features to be added (or even removed) from the stored graph state while inference is actively running. This would allow each embedding request to reflect an up-to-date view of the graph, improving the quality and responsiveness of predictions. Enabling such updates raises the need for consistent and efficient state modification.

Flink's current state backend is tightly integrated into its execution engine, which limits flexibility in managing graph state. A possible extension is to decouple the graph state from Flink altogether by integrating with an external graph database. Using a system like KuzuDB [12, 18, 22], or similar would eliminate the need to manually manage graph-to-keyed-state mappings and could provide richer graph query capabilities. This separation would also make it easier to share the graph across multiple jobs or services, rather than duplicating state across different operator UIDs in Flink.

Further optimizations within Flink could explore advanced streaming graph partitioning techniques [1, 26] to improve data locality and mitigate the communication overhead inherent in distributed traversals (Challenge 2). Alternatively, investigating these GNN integration challenges using different stream processing engines [10], which may offer distinct design trade-offs, could also yield valuable insights. Our system lays the groundwork for integrating GNN inference into stream processing architectures, but scaling it to more dynamic, large-scale, and adaptive deployments opens up many promising avenues for future research.

8 Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work was supported by the National Science Foundation under Grant No. 2237193 and 2440334.

References

[1] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming graph partitioning: an experimental study. *Proc. VLDB Endow.* 11, 11 (July 2018), 1590–1603. doi:10.14778/3236187.3236208

[2] Tyler Akidau, Edmon Begoli, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. 2021. Watermarks in stream processing systems: semantics and comparative analysis of Apache Flink and Google cloud dataflow. *Proc. VLDB Endow.* 14, 12 (July 2021), 3135–3147. doi:10.14778/3476311.3476389

[3] Amazon Web Services AI Shanghai Lablet (ASAIL), Amazon Web Services Machine Learning, NVIDIA, New York University, Georgia Institute of Technology. 2025. DGL: Deep Graph Library. https://www.dgl.ai/.

[4] Apache Flink. 2024. Asynchronous I/O. https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/dev/datastream/operators/asyncio/. Accessed: 2025-04-04.

[5] Apache Flink. 2024. Savepoints - Apache Flink Documentation (Release 1.20). https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/ops/state/savepoints/. Accessed: 2025-04-04.

[6] Apache Flink. 2024. Time Characteristics in Flink: Event Time, Processing Time, and Ingestion Time. https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/concepts/time/. Accessed: 2025-04-04.

[7] Jason Baumgartner. 2015. Pushshift Reddit Dataset. https://pushshift.io/.

[8] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1718–1729. doi:10.14778/3137765.3137777

[9] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rytstxWAW

[10] Luca De Martini, Alessandro Margara, and Gianpaolo Cugola. 2022. Analysis of market data with Noir: DEBS grand challenge. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems* (Copenhagen, Denmark) *(DEBS '22)*. Association for Computing Machinery, New York, NY, USA, 139–144. doi:10.1145/3524860.3539646

[11] Songgaojun Deng, Huzefa Rangwala, and Yue Ning. 2019. Learning Dynamic Context Graphs for Predicting Social Events. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 1007–1016. doi:10.1145/3292500.3330919

[12] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. Kùzu graph database management system. In *The Conference on Innovative Data Systems Research*, Vol. 7. 25–35.

[13] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2023. A survey on the evolution of stream processing systems. *The VLDB Journal* 33, 2 (Nov. 2023), 507–541. doi:10.1007/s00778-023-00819-8

[14] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[15] Sonia-Florina Horchidan, Po Hao Chen, Emmanouil Kritharakis, Paris Carbone, and Vasiliki Kalavri. 2024. Crayfish: Navigating the Labyrinth of Machine Learning Inference in Stream Processing Systems. In *Advances in Database Technology - EDBT : (Advances in Database Technology - EDBT, Vol. 27)*. Open Proceedings.org, Article 3, 676–689 pages. doi:10.48786/edbt.2024.58 QC 20240507Part of ISBN:978-389318091-2, 978-389318094-3, 978-389318095-0.

[16] Haichuan Hu, Fangming Liu, Qiangyu Pei, Yongjie Yuan, Zichen Xu, and Lin Wang. 2024. Grapher: A Resource-Efficient Serverless System for GNN Serving through Graph Sharing. In *Proceedings of the ACM Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 2826–2835. doi:10.1145/3589334.3645383

[17] Vasiliki Kalavri and John Liagouris. 2020. In support of workload-aware streaming state management. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.

[18] KuzuDB Team. 2024. KuzuDB: An Embedded Graph Database. https://kuzudb.com.

[19] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. 2018. Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SJiHXGWAZ

[20] Yang Liu, Xiang Ao, Zidi Qin, Jianfeng Chi, Jinghua Feng, Hao Yang, and Qing He. 2021. Pick and Choose: A GNN-based Imbalanced Learning Approach for Fraud Detection. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 3168–3177. doi:10.1145/3442381.3449989

[21] Yu-Chen Lo, Stefano E. Rensi, Wen Torng, and Russ B. Altman. 2018. Machine learning in chemoinformatics and drug discovery. *Drug Discovery Today* 23, 8 (2018), 1538–1546. doi:10.1016/j.drudis.2018.05.010

[22] Semih Salihoglu. 2023. Kùzu: A Database Management System For "Beyond Relational" Workloads. *SIGMOD Rec.* 52, 3 (Nov. 2023), 39–40. doi:10.1145/3631504.3631514

[23] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. 2015. An Overview of Microsoft Academic Service (MAS) and Applications. In *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy) *(WWW '15 Companion)*. Association for Computing Machinery, New York, NY, USA, 243–246. doi:10.1145/2740908.2742839

[24] Yuhang Song, Po Hao Chen, Yuchen Lu, Naima Abrar, and Vasiliki Kalavri. 2024. In situ neighborhood sampling for large-scale GNN training. In *Proceedings of the 20th International Workshop on Data Management on New Hardware* (Santiago, AA, Chile) *(DaMoN '24)*. Association for Computing Machinery, New York, NY, USA, Article 11, 5 pages. doi:10.1145/3662010.3663443

[25] Apache Spark. 2023. Structured Streaming Programming Guide. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

[26] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Beijing, China) *(KDD '12)*. Association for Computing Machinery, New York, NY, USA, 1222–1230. doi:10.1145/2339530.2339722

[27] Jie Sun, Zuocheng Shi, Li Su, Wenting Shen, Zeke Wang, Yong Li, Wenyuan Yu, Wei Lin, Fei Wu, Bingsheng He, and Jingren Zhou. 2025. Helios: Efficient Distributed Dynamic Graph Sampling for Online GNN Inference. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, USA) *(PPoPP '25)*. Association for Computing Machinery, New York, NY, USA, 2–15. doi:10.1145/3710848.3710854

[28] Zeyuan Tan, Xiulong Yuan, Congjie He, Man-Kit Sit, Guo Li, and Xiaoze Liu. 2023. Baole Ai, Kai Zeng, Peter Pietzuch, and Luo Mai. 2023. Quiver: Supporting GPUs for Low-Latency, High-Throughput GNN Serving with Workload Awareness. *arXiv preprint arXiv:2305.10863* (2023).

[29] The Apache Software Foundation. [n. d.]. Apache Flink. https://flink.apache.org/. Accessed: January 2024.

[30] The Apache Software Foundation. 2024. Apache Storm. https://storm.apache.org/. Accessed: January 2024.

[31] Dan Wu, Zhaoying Li, and Tulika Mitra. 2023. InkStream: Real-time GNN Inference on Streaming Graphs via Incremental Update. arXiv:2309.11071 [cs.LG] https://arxiv.org/abs/2309.11071

[32] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2022. Graph Neural Networks in Recommender Systems: A Survey. *ACM Comput. Surv.* 55, 5, Article 97 (Dec. 2022), 37 pages. doi:10.1145/3535101

[33] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021), 4–24. doi:10.1109/TNNLS.2020.2978386

[34] Jason Yik, Sanmukh R. Kuppannagari, Hanqing Zeng, and Viktor K. Prasanna. 2022. Input Feature Pruning for Accelerating GNN Inference on Heterogeneous Platforms. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 282–291. doi:10.1109/HiPC56025.2022.00045

[35] Peiqi Yin, Xiao Yan, Jinjing Zhou, Qiang Fu, Zhenkun Cai, James Cheng, Bo Tang, and Minjie Wang. 2023. DGI: An Easy and Efficient Framework for GNN Model Evaluation. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 5439–5450. doi:10.1145/3580305.3599805

[36] Peiqi Yin, Xiao Yan, Jinjing Zhou, Qiang Fu, Zhenkun Cai, James Cheng, Bo Tang, and Minjie Wang. 2023. DGI: An Easy and Efficient Framework for GNN Model Evaluation. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 5439–5450. doi:10.1145/3580305.3599805

[37] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 974–983. doi:10.1145/3219819.3219890

[38] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 974–983. doi:10.1145/3219819.3219890

[39] Haochen Yuan, Yuanqing Wang, Wenhao Xie, Yu Cheng, Ziming Miao, Lingxiao Ma, Jilong Xue, and Zhi Yang. 2025. NeuStream: Bridging Deep Learning Serving and Stream Processing. In *Proceedings of the Twentieth European Conference on Computer Systems*. 671–685.

[40] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs . In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE Computer Society, Los Alamitos, CA, USA, 36–44. doi:10.1109/IA351965.2020.00011

[41] Hongkuan Zhou, Ajitesh Srivastava, Hanqing Zeng, Rajgopal Kannan, and Viktor Prasanna. 2021. Accelerating large scale real-time GNN inference using channel pruning. *Proc. VLDB Endow.* 14, 9 (May 2021), 1597–1605. doi:10.14778/3461535.3461547

[42] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph Neural Networks: A Review of Methods and Applications. *AI Open* 1 (2020), 57–81. https://arxiv.org/abs/1812.08434