RESEARCH-ARTICLE

FaaSGNN: Enabling Memory Efficient and Low Latency GNN Inference Services with Serverless Computing

YUZHUO YANG, Shanghai Jiao Tong University, Shanghai, China

KAIHUA FU, Shanghai Jiao Tong University, Shanghai, China

QUAN CHEN, Shanghai Jiao Tong University, Shanghai, China

DEZE ZENG, China University of Geosciences, Wuhan, Hubei, China

SHUO QUAN, China Telecom Corporation Limited, Beijing, Beijing, China

JIE WU, Temple University, Philadelphia, PA, United States

View all

Published: 19 November 2025

SoCC '25: ACM Symposium on Cloud Computing
November 19 - 21, 2025
Online, USA

Conference Sponsors:
SIGOPS
SIGMOD

https://doi.org/10.1145/3772052.3772253
ISBN: 9798400722769

FaaSGNN: Enabling Memory Efficient and Low Latency GNN Inference Services with Serverless Computing

Yuzhuo Yang
Shanghai Jiao Tong University
Shanghai, China
yang-yz@sjtu.edu.cn

Kaihua Fu
Shanghai Jiao Tong University
Shanghai, China
midway@sjtu.edu.cn

Quan Chen
Shanghai Jiao Tong University
Shanghai, China
chen-quan@cs.sjtu.edu.cn

Deze Zeng
China University of Geosciences
Wuhan, China
deze@cug.edu.cn

Shuo Quan
Cloud Computing Research Institute,
China Telecom
Beijing, China
quansh@chinatelecom.cn

Jie Wu
Temple University
Philadelphia, United States
jiewu@temple.edu

Minyi Guo
Shanghai Jiao Tong University
Shanghai, China
guo-my@cs.sjtu.edu.cn

Abstract

While GNN-based services often experience load fluctuation, applying serverless computing to serve GNN inference reduces the cost and allows elastic resource scaling. However, GNN serverless shows poor performance due to heavy data fetching latency and long cold startup overhead, and our observation indicates opportunities for reducing data redundancy and mitigating cold startup latency. In this paper, we present **FaaSGNN**, a serverless GNN inference framework that enables low latency and memory efficient GNN serving through three key designs: (i) *serverless-native on-demand graph fetching strategy* that enables lightweight in-container graph sampling with full dataset resides in remote; (ii) *memory-aware adaptive feature caching policy*, which facilitates data reuse between requests to reduce redundant fetching; and (iii) *load-aware request scheduler*, which reschedules requests to bypass cold start and achieve load balance between containers. Experimental results show that FaaSGNN achieves a 5.6× lower end-to-end latency and 57.1% less memory usage on average compared to state-of-the-art works.

CCS Concepts

• **Computer systems organization → Cloud computing**.

Keywords

Serverless Computing, Graph Neural Network

SoCC '25, Online, USA

ACM ISBN 979-8-4007-2276-9/2025/11
https://doi.org/10.1145/3772052.3772253

ACM Reference Format:
Yuzhuo Yang, Kaihua Fu, Quan Chen, Deze Zeng, Shuo Quan, Jie Wu, and Minyi Guo. 2025. FaaSGNN: Enabling Memory Efficient and Low Latency GNN Inference Services with Serverless Computing. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3772052.3772253

1 Introduction

Graph Neural Networks (GNNs) [22, 28, 55, 63] have emerged as a paradigm for learning representations from graph structured data and have achieved impressive performance in many graph-related tasks, e.g. node classification [17]. By witnessing the success of GNNs, many cloud applications have incorporated GNNs in their service, such as e-commerce [45, 64], recommendation [62, 66], financial fraud detection [39, 57], and social network analysis [32]. While user facing cloud applications/services often experience diurnal load patterns [38, 52, 67], GNN-based services also experience dynamic loads [24]. In this case, for the low cost and quick resource scale, serverless computing [34] that scales resources using lightweight containers based on the real-time load is used to handle the spikes [11, 12, 31]. According to AWS's cost model, serverless computing-based GNN service reduces the cost by 48.6%, compared with long term deployment.

Figure 1 shows the workflow of an example GNN inference service, consisting of a sample phase and an inference phase. As observed, to process a GNN query, a multi-hop neighbor subgraph of the request's query vertex is first extracted, then features corresponding to sampled subgraph are collected, followed by executing GNN model to derive the desired output. When deployed with serverless paradigm, both sample and inference phases are executed in containers with data stored in remote database. Existing works typically adopt *full-data fetching scheme* [58] and *on-demand fetching scheme* [72] for accessing remote data, while following *naive policy* [10] for resource scaling. Although prior works successfully realize serverless GNN serving with elastic scaling, they

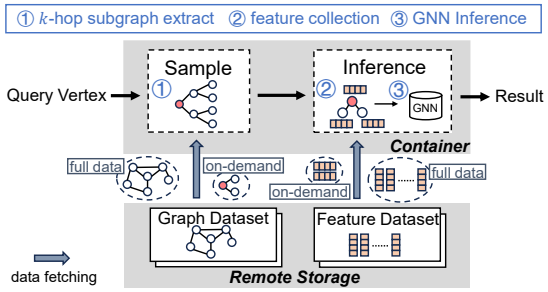


Figure 1: The execution flow of a GNN inference service.

still suffer from long response latency and high memory overhead due to the following two main issues:

(i) **Redundant data fetching**. For *full-data fetching scheme* [58], the entire graph topology and feature data are fetched from remote during container cold start. However, as our analysis shows that the actual size of data required by most inference requests is less than 10% of entire dataset (Fig. 5, Section 3.4), such scheme causes large memory usage and long fetching duration. Worse, the long fetching latency can result in more container initialization when the load spikes, further increasing the memory consumption. For *on-demand fetching scheme* [72], only the subgraph and associated features required by requests are fetched into container. Although such a scheme reduces data accessing volume during cold start compared with full-data fetching scheme, it further introduces data fetching in warm containers as each request requires different on-demand data, and our investigation indicates that there exists *neighbor vertex overlap* between requests, with the overlap rate exceeding 75% for most requests in popular real-world traces of GNN service (Section 3.4).

(ii) **Cold startup incurred load imbalance**. For *naive scaling policy*, a request triggers initialization of a new container when there is no idle container available upon arrival, and waits for processing until the new container is ready. While the policy works well for traditional serverless computing workload where launching container is lightweight, it shows limitation when handling GNN inference service that requires complex runtime environment (e.g., GNN frameworks DGL [58] and PyG [19]). Concretely, the cold start of complex GNN runtime suffers from long latency (more than 5 second in our test), which may even exceed the processing time of requests. Due to such latency, adopting naive scaling policy for GNN workload brings load imbalance, as some containers may become idle by completing the previous request, while some requests are still waiting for containers undergoing cold start.

To address the aforementioned limitations appears to be nontrivial. While it is promising to enable on-demand fetching and utilize data overlap for reducing overhead incurred by redundant fetching both in cold and warm containers, several challenges remain unresolved by existing works. Firstly, serverless computing typically supports fast scaling and complete recycle of all computing instances (namely *scaling to zero*) to efficiently handle dynamic loads. However, existing on-demand fetching implementation [72] employs long-term graph database service, which is not cost-efficient under serverless paradigm due to its inability of fast scaling and scaling to zero. Secondly, the limited container memory capacity greatly suppresses the effectiveness of traditional static caching policy [65], making it difficult to fully leverage data overlap without additional memory cost. Besides, for load imbalance incurred by large latency of initializing GNN runtime, current cold startup optimization [13, 18, 23] turns out to be less effective for this issue. It remains challenging to redesign scaling policy for mitigating cold startup latency, with an emphasis on ensuring efficient scaling.

This paper offers the following insights to help resolve the above challenges. 1) It is possible to partially fetch the required data directly from remote storage on demand in containers, if the graph topology and features are stored in the Compressed Sparse Row (CSR) and vector formats. This is because CSR supports efficient graph traversal through simple index lookup as each vertex's adjacent list is stored in contiguous space, which can also be easily parallelized. 2) The memory usage during inference (the features) can be accurately calculated based on the sampled subgraph, though the required memory space of a request's subgraph is not known beforehand. In this case, it is possible to efficiently cache features by leveraging free memory space in containers, which can be determined based on the memory usage information during inference. However, caching graph topology for sampling could incur memory insufficiency due to lack of memory usage information.

Based on our analysis and insights, we propose a novel serverless GNN inference framework named **FaaSGNN**. FaaSGNN comprises a *serverless-native on-demand graph fetching* strategy, a *memory-aware adaptive feature caching* policy and a *load-aware request scheduler*. Enabling the serverless-native on-demand fetching, the subgraph of a request is iteratively fetched from binary graph data in remote storage with low latency and memory consumption. In each container, a feature cache is used to preserve the feature data of requests after execution so that a new request only needs to fetch the non-cached features. Considering the limited memory of a container, the caching policy evicts cold features when detecting insufficient memory for execution. The request scheduler determines whether to launch new containers for incoming requests according to the current load, and schedules requests to queues associated with warm containers. Once the scheduler detects idle containers or successfully started new containers, it dynamically reschedules the queued requests to the idle container for load balancing.

We evaluate FaaSGNN using two popular GNNs, i.e. GCN [28] and GAT [55], on four real-word graph datasets [8, 29] with Microsoft Azure Function traces [50]. Experimental results show that FaaSGNN achieves 5.6× lower end-to-end latency and 57.1% less memory usage compared to state-of-the-art works.

Overall, the main contributions of our paper can be summarized as follows:

- We identify the inefficiencies of GNN inference service with serverless computing. Our analysis further motivates the design of FaaSGNN.
- We propose *serverless native on-demand data fetching* strategy and *adaptive feature caching* policy that effectively reduce the redundant data fetching both in cold and warm containers, without requiring long-term backend service or incurring additional memory cost for caching.
- We propose a *load-aware request scheduler* that enables a brand new scaling policy with dynamic scheduling to effectively

mitigate load imbalance, while ensuring the throughput of the whole system.
- We demonstrate the superiority of FaaSGNN through extensive experiments, which are conducted by leveraging real-world graph and trace datasets at different scales for providing comprehensive evaluation at various dimensions.

2 Related Work

GNN training and inference. Extensive GNN architectures have been proposed by existing works [22, 28, 44, 53, 55, 63, 69] to facilitate machine learning on graphs. To reduce redundant computation, many works [14, 15, 22, 25, 70] construct graph samples for GNN training and inference.

Prior works have proposed many GNN systems to improve inference and training efficiency. For training, some works [59–61] develop systems to accelerate GNN operators, while other works [37, 40, 65] extend GNN training to distributed environment. Dorylus [54] is the typical system that leverages serverless computing for GNN training. As for inference, some architectures for accelerating inference on specific hardware are proposed [42, 48]. In terms of serving GNN inference, GraphLearn [74] introduces an online GNN inference service, which is not deployed with serverless paradigm.

ML model inference with serverless. Serving machine learning (ML) models with serverless computing has been widely explored in recent years [11, 16, 31, 67, 71]. BATCH [11], MArk [71] and INFless [67] enable request batching to enhance throughput without violating SLO, and other works [16, 31] leverage tensor sharing to improve performance. Besides, ServerlessLLM [20] utilizes data locality to reduce inference latency. There also exist industrial implementations, including AWS SageMaker [7] and Azure ML [1], with KServe [5] being an open-source initiative. However, above works mainly deal with traditional DNN inference intended for tasks related to NLP or vision, and may encounter inefficiency when handling GNN inference. λGrapher [24] is a relevant GNN system that aims to reduce computing resource usage (e.g. CPU cores, memory) under SLO configuration. Although it shows less resource consumption by adopting batch processing with subgraph-centric scheduling, its designs will lose effectiveness when deployed under First-Come-First-Serve(FCFS) paradigm which is commonly practiced by existing serverless platforms [10]. This is because the resource usage cannot be amortized across requests under FCFS, and no specific optimization is proposed for reducing execution latency of individual request in λGrapher. Moreover, it ignores overhead incurred by remote storage and container initialization that are key features in serverless computing and potentially affect performance. This further imposes limitations on the effectiveness of λGrapher. Overall, it still remains challenging to combine GNN inference with serverless.

Cold startup optimization. Cold startup latency in serverless computing is non-negligible. Recent efforts to mitigate cold startup latency can be divided into following three categories: (i) image fetching optimization [23, 30, 73], (ii) container caching [13, 36, 46, 68] and (iii) container checkpoint [18, 51]. For image fetching, Slacker [23] builds a docker storage driver that achieves fast image distribution, Wharf [73] reduces the image synchronization overhead, and DADI [30] designs a block-based image service for efficient image fetching. However, the overhead of container instance initialization is not considered by these works. For container caching, SEUSS [13], FlashCube [36] and RainbowCake [68] construct containers from cached snapshots or pre-warmed container parts, while IceBreaker [46] dynamically determines the scale of function pre-warming. Nevertheless, using in-memory cache for containers incurs additional memory consumption apart from function execution. For container checkpoint, Catalyzer [18] and Prebaking [51] launch containers from restored checkpoint of function instance. However, launching a container of GNN inference from checkpoint still incurs long latency due to the time of initializing the large runtime environment (only reduced from 5s to 4s in our test). This indicates that serverless GNN inference service needs a method to effectively mitigate cold startup overheads without using additional memory.

In-memory caching mechanism. Exploiting resources for enabling in-memory caching is demonstrated to be effective at reducing latency incurred by data movement. For GNN training, GNNLab [65] uses a static caching policy to accelerate data access. It profiles peak memory usage during training, and thereby determines the remaining memory space for cache which remains unchanged through the entire process. However, the effectiveness of such a design will be greatly suppressed under serverless computing, since the limited container memory capacity significantly reduces the static cache size. For caching with serverless, InfiniCache [56] develops an in-memory caching service using serverless functions. Nevertheless, it incurs additional memory consumption as an independent service which results in low cost efficiency. OFC [43] dynamically identifies over-provisioned memory with prediction model and develops in-memory cache by utilizing that memory. Although it achieves effective caching without additional memory, it only supports coarse-grained memory predictions for traditional data types (e.g. image). When deployed for GNN workloads, the input parameters of the prediction model require re-identification. Moreover, the caching workflow requires adjustments since the memory usage information of a request cannot be captured before acquiring corresponding subgraph. Therefore, serverless GNN inference needs a complementary caching mechanism, with fine-grained memory usage prediction and dedicated design of caching strategy tailored to GNN inference to accelerate data access.

3 Background and Motivation

In this section, we present the poor performance of prior implementations of serverless GNN inference, identify the root causes of the inefficiency, and discuss the opportunities and challenges to optimize serverless GNN inference.

3.1 Graph Neural Networks

GNNs show strong capability for graph-based machine learning. A graph $G = (V, E)$ is a tuple of vertex set V and edge set E, and each vertex $v \in V$ is associated with a feature vector with d dimensions. Prevalent GNN models [22, 28, 55, 63, 69] comprise multiple layers and each layer computes feature vector for vertex v by two main operators: aggregate and update. The "aggregate" operator

collects features of neighbors of vertex v and generates intermediate representations. Then the "update" operator, which is generally composed of standard neural network operations like matrix multiplication, generates the output from intermediate vector. Equation 1 illustrates the detailed computation of these two operators:

$$\begin{aligned} \hat{h}_i^{(l)} &= \text{aggregate}^{(l)}(\{h_j^{(l-1)} | v_j \in N(v_i)\}) \\ &= c_{ii}^{(l)} h_i^{(l-1)} + \sum_{v_j \in N(v_i)} c_{ij}^{(l)} h_j^{(l-1)} \\ h_i^{(l)} &= \text{update}^{(l)}(\hat{h}_i^{(l)}) \\ &= \sigma(W^{(l)} \hat{h}_i^{(l)}), \end{aligned} \tag{1}$$

where $h_i^{(l)} \in \mathbf{R}^{d^{(l)}}$ is the hidden representation of vertex v_i after the l-th layer, $N(v_i)$ denotes the neighbors of vertex v_i, c_{ii} and c_{ij} denote the weights for aggregation in different GNNs, $W^{(l)} \in \mathbf{R}^{d^{(l)} \times d^{(l-1)}}$ and σ represent the matrix for feature transformation and non-linear activation function respectively. For instance, in GCN [28], the aggregation weights are defined as $c_{ij} = (\deg(v_i) \cdot \deg(v_j))^{-1/2}$ and $c_{ii} = (\deg(v_i))^{-1}$ where $\deg(\cdot)$ denotes the degree of the vertex.

Sample-based GNN inference: When the number of GNN layers is k, processing request with query vertex v only requires the k-hop neighbors' features of v. To reduce redundant computation, existing works adopt sample-based GNN inference [74]. Sample-based method first extracts the subgraph consisting of k-hop neighbors of query vertex v, then performs "aggregate" and "update" operators on this sampled subgraph. The value of k is commonly set to 2 in popular GNN models [28, 55].

3.2 GNN Inference with Serverless Computing

Serverless computing [26, 27, 34, 49] offers high elasticity with lightweight containers and a pay-as-you-go billing. Upon arrival of the request, a cold container is started if no container is available [10]. After processing requests, warm containers are kept alive for a period to serve potential incoming requests [21, 33]. For execution of GNN inference request under serverless computing, the container needs to fetch corresponding graph and feature data of the request from remote storage, and then performs inference. As presented in Section 1, current GNN serverless implementations for data fetching can be categorized into two schemes, i.e. full-data fetching [58] and on-demand fetching [72]. *DGL-serverless* that directly enables GNN framework DGL [58] in serverless is the representative implementation of the full-data fetching. *AWSGNN* [72] is a representative implementation of the on-demand fetching. With both implementations, the memory capacity of a container is set to the peak memory usage during processing requests, which is determined by the memory usage of processing vertex with maximum in-degree for each dataset. This guarantees sufficient memory for processing each vertex.

3.3 Inefficiency of Existing Solutions

We first investigate the efficiency of *DGL-serverless* and *AWSGNN*, in terms of end-to-end latency and memory consumption. Specifically, we use three representative real-world graph datasets with vertex ID sequences representing query vertices: *reddit-hyperlink* (RE), *higgs-twitter* (TW) and *stackoverflow* (ST) [29] to perform the

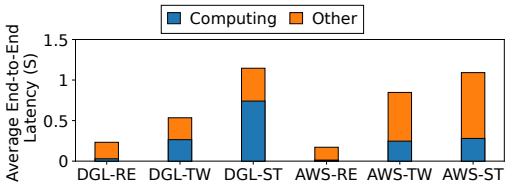


Figure 2: The average end-to-end latency of DGL-serverless and AWSGNN on different datasets.

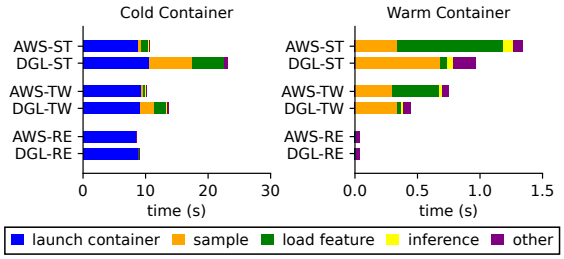


Figure 3: End-to-end request execution time breakdown in cold and warm containers.

investigation. The popular 2-layer GCN [28] model is used to run the datasets. Other models show similar results. Since there is no public dataset that includes the request arrival pattern, we refer to Microsoft Azure Function trace (MAF) [50] to generate requests without loss of generality. Table 1 in Section 5.1 summarizes the detailed hardware and software configuration. Both *DGL-serverless* and *AWSGNN* independently scale sample and inference containers for better resource utilization, since the two phases exhibit highly distinct resource demands to support the same throughput.

Figure 2 shows the average end-to-end latency of DGL-serverless and AWSGNN on the three benchmarks. In this figure, the x axis shows the experimental combinations. For example, "DGL-RE" and "AWS-RE" represent the experiment that runs the benchmark "RE" with DGL-serverless and AWSGNN respectively. As shown, the GNN computation latency in sample and inference phases only accounts for 42.2%/20.7% on average (up to 64.6%/29.1%) with DGL-serverless and AWSGNN. The extra overheads of both implementations are non-negligible. To further investigate the overhead, we randomly select one vertex ID per dataset and test end-to-end latency in cold and warm containers.

Figure 3 shows the detailed end-to-end latency breakdown. For DGL-serverless, warm containers perform well, but cold containers suffer from large cold startup and data fetching latency, which account for 69.4% and 28.5% of the latency on average, respectively. This is because DGL-serverless initializes a complex execution environment and fetches the complete graph topology and features during cold startup. For AWSGNN, container launch still dominates overhead in cold containers, while data fetching becomes the main overhead in warm containers (55.3% on average). This is because AWSGNN fetches the neighbor subgraph and corresponding features of the query vertex for each request before inference, and such data fetching is not required by DGL-serverless in warm containers.

Since memory usage impacts the number of containers that can be deployed in a node, we further investigate the total memory

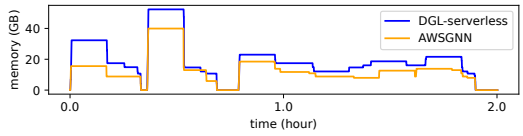
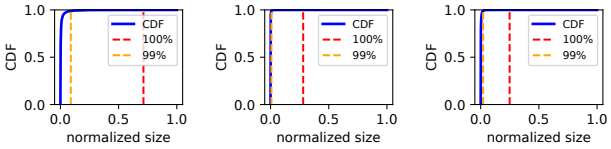


Figure 4: Memory usage of the benchmark TW with DGL-serverless and AWSGNN.



(a) RE (b) TW (c) ST

Figure 5: CDF of the size of a 2-hop subgraph.

usage of DGL-serverless and AWSGNN. In the experiment, we extract a period from the trace of the function which has the highest number of invocations in MAF trace dataset to simulate the arrival pattern of requests. Figure 4 shows the timeline of memory usage on TW dataset. As observed, DGL-serverless consumes more memory than AWSGNN, as DGL-serverless configures containers with higher memory capacity for preserving the entire dataset in memory. Moreover, when the load of requests spikes, DGL-serverless tends to launch more containers due to its long data fetching latency in cold containers, leading to substantial memory allocation and causing memory waste regarding the keep-alive policy.

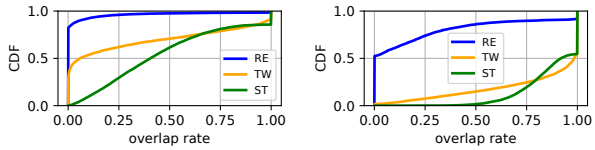
To conclude, DGL-serverless suffers from long response latency during container cold start, and AWSGNN suffers from long response latency even with warm container. DGL-serverless also suffers from high memory usage that results in high monetary cost.

3.4 Root Causes of the Inefficiency

The inefficiency originates from *redundant data fetching* and *cold startup incurred load imbalance.*

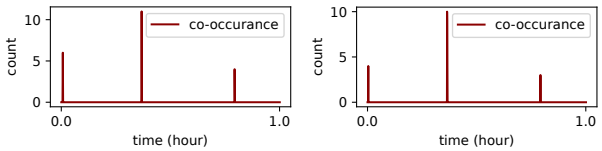
3.4.1 Redundant Data Fetching. A 2-layer GNN requires sampling of 2-hop subgraph, and Figure 5 shows the cumulative distribution function (CDF) of vertex's 2-hop subgraph size in three datasets normalized to the full graph size. As observed, for TW and ST, a vertex's 2-hop subgraph occupies at most 30% of the full graph, with 99% of vertices accounting for less than 5%. For the smallest dataset RE, the largest 2-hop neighbor subgraph of a vertex is 70% of the full graph size, with 99% vertices having subgraph size less than 10% of full graph. Despite the small amount of required data, DGL-serverless fetches the full graph topology and features in cold containers. This results in both long data loading time and large memory usage. Though AWSGNN reduces the data fetching time in cold containers by adopting on-demand fetching scheme, every request in warm containers needs to fetch the required data, as the neighbor subgraph of query vertex differs across different requests. Such data fetching introduces latency within warm containers.

After carefully analyzing existing traces, we find that there are data overlaps in neighbor subgraph across different requests. Figure 6 shows the CDF of the vertex overlap rate of each request with



(a) preceding 1 request (b) preceding 10 requests

Figure 6: CDF of subgraph overlap rate.



(a) DGL-serverless (b) AWSGNN

Figure 7: Co-occurance of idle and cold start containers.

its prior 1 request and 10 requests in AWSGNN. As observed, for reusing data from prior 1 request, more than 40% requests have more than 50% overlap in TW and ST, while RE shows less overlap. For reusing data from prior 10 requests, over 75% of requests exhibit more than 75% overlap in TW and ST, while 25% of requests in RE have more than 25% overlap.

However, current solutions fail to leverage the overlap, and require each request to load the data again, even if the topology and feature data of a request's neighbor subgraph have already been partially loaded into memory during serving prior requests. If the overlapped data from completed requests can be utilized, the data fetching latency in warm containers can be reduced, and reusing more historical data can further reduce redundant fetching.

3.4.2 Cold Startup Incurred Load Imbalance. Existing scaling policy adopted by most serverless frameworks launches a new container when a request arrives with no pre-initialized container being available [10], and processes the request in the newly launched container after its initialization. Such a policy performs poorly for GNN workloads, where functions require initialization of complex GNN runtime (e.g. PyTorch and DGL libraries, totaling up to 2.5GB). As a result, container cold startups experience high latency, often exceeding the request execution time. Such latency can cause load imbalance and hinder performance. This is because when some requests are waiting for initialization of cold containers which they are scheduled to, some warm containers could possibly finish processing prior requests and become idle.

In particular, the load imbalance can be exacerbated when the request load spikes, as a large number of new containers are initialized and allocated to requests. Such imbalance could be measured by the co-occurrence value, which is defined as the minimum number of idle and cold containers at a certain timestamp. The positive co-occurrence value indicates that there are idle containers available while some requests are waiting for cold startup containers. Figure 7 reflects such imbalance when serving requests from TW dataset following the arrival pattern in MAF traces, which exhibits three spikes in one hour. The high value of co-occurrence indicates that efficiently scheduling requests waiting for cold startups to idle

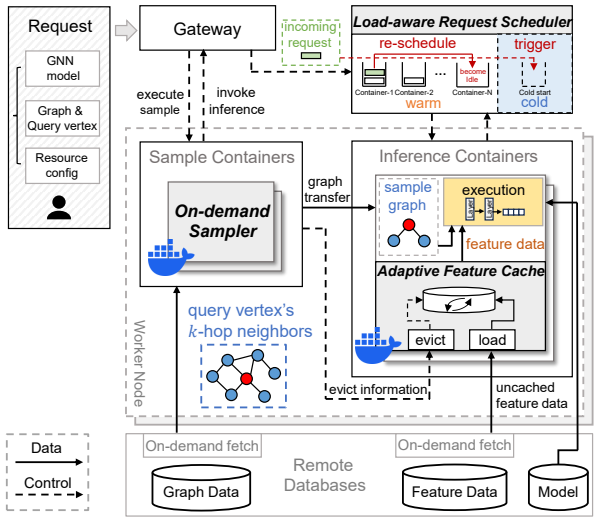


Figure 8: Overview of FaaSGNN.

containers could help bypass the latency associated with container initialization.

4 Design of FaaSGNN

In this section, we first present the design overview of FaaSGNN, then elaborate on the core designs of FaaSGNN.

4.1 Overview of FaaSGNN

In order to resolve the aforementioned inefficiency, we propose a novel serverless GNN inference framework named FaaSGNN, which comprises three core designs: *serverless-native on-demand graph fetching* strategy, *memory-aware adaptive feature caching* policy and *load-aware request scheduler*. In FaaSGNN, each sample container employs an on-demand strategy to fetch neighbor subgraph of query vertex from remote storage to mitigate fetching latency in cold container. For utilization of data overlap, FaaSGNN maintains an adaptive feature cache in each inference container, which stores frequently accessed features to reduce data fetching overhead in warm containers. To further balance the load and mitigate cold start overhead, FaaSGNN incorporates a load-aware request scheduler to dynamically re-schedule incoming requests to idle containers, while it also controls container scaling for resource efficiency. Figure 8 shows the design overview of FaaSGNN.

To effectively leverage the opportunities as analyzed and achieve cost efficiency, the designs of different components of FaaSGNN address specific challenges that remain unresolved by existing frameworks. For on-demand graph fetching, the challenging part lies in enabling the on-demand fetching without employing long-term backend services (e.g. graph database), which exhibit poor cost efficiency under serverless paradigm due to inability of fast scaling and scaling to zero. FaaSGNN organizes the graph data into row and column indices based on the Compressed Sparse Row (CSR) format, and stores these indices in two binary files for neighbor information lookup, in order to enable on-demand fetching in container. Each index is composed of several integers representing either vertex IDs or position indicators, which are stored in binary in consecutive memory and can be accessed through offset. During subgraph fetching, the required neighbor information is iteratively fetched, with only the demanded data being fetched into memory (Section 4.2).

For utilizing data overlap, the challenging part is managing the size of cache within limited container memory in order to avoid interfering with request execution. Meanwhile, it is also crucial to determine the cached data for maximizing data reuse. FaaSGNN introduces adaptive feature caching policy that preserves feature data of previous requests in memory after execution. With this design, only the non-cached features are fetched when processing requests. To better manage cache size and increase cache hit rate, a lightweight model is trained to predict request's memory usage during inference, and degree-based cache eviction happens when insufficient memory is detected for execution (Section 4.3).

For balancing load and controlling container scaling, the challenging part resides in determining when to launch new containers and dynamically scheduling incoming requests to warm containers. The load-aware scheduler maintains a request queue for each warm container to facilitate dynamic scheduling. Upon user request arrival, the scheduler first tries to allocate an idle container for execution. In case no idle container is available, the scheduler enables the request to queue for a warm container, and the request gets processed once prior executions are complete, thus bypassing cold startup latency. When detecting idle containers, requests waiting in queues will be re-scheduled for immediate execution, further achieving load balance (Section 4.4).

FaaSGNN works in the following steps for processing each request: 1) It allocates sample container and fetches neighbor subgraph of query vertex through on-demand graph fetching strategy, with inference container being allocated after sampling. 2) FaaSGNN estimates inference memory usage based on neighbor subgraph information using prediction model, and evicts several cache entries in inference container if needed. 3) Inference container receives neighbor subgraph from sample container and fetches features that are not in cache from remote. 4) Inference container collects required features from cache and executes GNN inference, then returns the result.

4.2 Serverless-native On-demand Fetching

Constructing neighbor subgraph with full graph topology residing in remote can be achieved through iteratively fetching demanded neighbor information of query vertex, so as to avoid unnecessary data fetching. For efficiently fetching neighbors given a specific vertex, FaaSGNN organizes the graph topology into two indices stored in binary form to support neighbor retrieval by offset.

Storage format. To enable efficient graph traversal and subgraph extraction, the graph storage format needs to facilitate fast access to adjacency information of each vertex. Traditional format, e.g. adjacency matrix, suffers from inefficient edge lookup which requires scanning the entire row for retrieving neighbors of a vertex. We therefore adopt CSR format which stores each vertex's adjacency information in contiguous space for fast graph traversal through simple index lookups. Each graph is represented by row index and column index in CSR format, with each index consisting

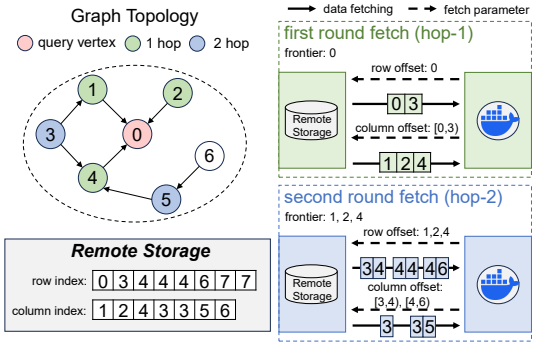


Figure 9: An example of serverless-native on-demand graph fetching strategy.

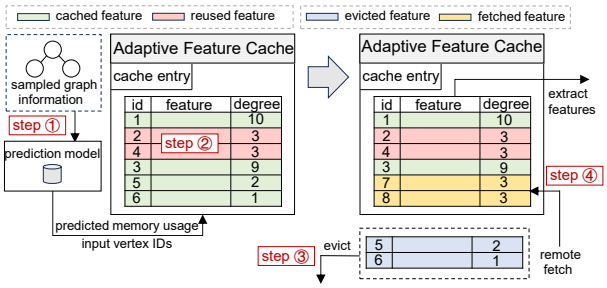


Figure 10: An example of adaptive feature cache.

of multiple integers that are stored in consecutive space of remote storage. The i-th and $(i+1)$-th elements in row index record the start and end positions of a segment in column index, with the elements in this segment representing the neighbors of the vertex with ID i. Therefore, accessing neighbors of a given vertex can be achieved through looking up two indices in order, and the corresponding part of data can be fetched based on offset. As for supporting dynamic update, we adopt the commonly practiced snapshot-based strategy, where the CSR indices of graphs are updated periodically and the latest snapshots of CSR indices are utilized by the service.

Graph Fetching. The neighbors of a vertex are fetched by following two steps. Firstly, the two consecutive integers with offset computed by the vertex ID is fetched from row index, which tell the segment storing neighbors of the vertex in column index. Secondly, the segment offset is computed by the fetched integers, and the neighbors of vertex are then fetched from column index.

When fetching neighbor subgraph of a given query vertex, FaaSGNN first initializes "frontier", which is a set of vertices whose neighbors are required to be fetched for the query vertex. Then it fetches neighbors for each vertex in frontier. After fetching, current frontier is replaced by newly fetched neighbors for next round of fetching. Note that vertices with neighbors being fetched in this round will not be processed in following procedure. FaaSGNN continues to execute several rounds of fetching, and terminates when the demanded neighbor subgraph has been fetched, with the number of rounds determined by number of GNN layers.

Figure 9 gives an example of graph remote storage format, and how on-demand graph fetching strategy fetches 2-hop subgraph for a query vertex. In first round of fetching, the frontier is set to query vertex 0, and the container fetches two integers from position 0 in row index. With the fetched integers indicating that the neighbors of vertex 0 are located in positions from 0 to 3, the container further fetches corresponding data from column index. In second round of fetching, the frontier is updated to previous fetched vertices 1, 2, and 4. The fetching process is similar to the first round, with container accessing row index and column index sequentially to fetch neighbors for vertices in frontier. After two rounds of fetching, the container obtains sufficient graph data and constructs 2-hop subgraph for query vertex.

As for on-demand feature fetching, the feature data that are associated with vertices in sampled subgraph are fetched through looking up feature table from remote after the on-demand sampler finishes subgraph fetching.

In addition, the proposed on-demand graph fetching strategy can be easily implemented in parallel. We implement the parallel on-demand graph fetching for FaaSGNN by configuring multiple threads to process index looking up and fetching for different vertices in frontier.

4.3 Memory-aware Adaptive Feature Caching

The data overlap between requests can be utilized through caching. Existing caching policy [65] reserves a fixed amount of memory for cache, which is determined by the rest of available memory after profiling the peak memory usage during execution. However, such caching policy suffers from poor performance when adopted in serverless containers with memory constraint. As the container memory capacity is configured slightly larger than peak memory usage during execution, little space remains for cache under static caching policy. Therefore, FaaSGNN introduces a memory-aware adaptive caching policy.

Adaptive caching. As our observation shows that the neighbor subgraph size of 99% vertices is much smaller than the maximum size, adaptive caching policy continues to cache fetched data of requests to make full use of free memory space when the memory usage during execution is low, and evicts cache entries to guarantee execution when detecting memory insufficiency.

The memory usage for processing GNN inference consists of the following parts: neighbor subgraph, feature data, and intermediate output of GNN model. It is feasible to estimate memory usage when information about the input data and GNN model architecture is available. However, as the size of neighbor subgraph remains unknown until sampling is complete, the memory usage during sampling is unpredictable. This poses obstacles for cache size management and hinders adaptive caching for graph topology data. Therefore, FaaSGNN adopts adaptive feature caching policy since all information of input data is available after sampling. Such design shows promise in improving latency as feature fetching typically dominates the total fetching latency.

As for caching feature data, each cache entry contains three parts: *vertex ID*, *feature data*, and *vertex degree* . The degree of each vertex is obtained during sampling and is leveraged for cache eviction, which is discussed later. For fetching features of a given neighbor subgraph with feature cache, FaaSGNN identifies vertices whose features are not in cache by checking vertex IDs, then fetches

associated features from remote storage into memory and updates the cache entries. Afterwards, features of neighbor subgraph are collected from cache and processed by GNN model.

Figure 10 depicts an example of the adaptive feature caching policy when processing a request. FaaSGNN first takes information of sampled neighbor subgraph as input and uses a prediction model to predict the peak memory usage during inference (step 1), then identifies features that are associated with current neighbor subgraph and features need to be fetched (step 2). After that, FaaSGNN computes the number of entries to be evicted, then performs cache eviction using a degree-based strategy where entries with the smallest degrees are evicted (step 3). Finally, uncached features that are associated with current neighbor subgraph are fetched into the cache, and FaaSGNN extracts features from the cache (step 4).

Memory usage prediction. To avoid interference with GNN execution, FaaSGNN introduces a prediction model which uses GNN input and model architecture information to predict the peak memory usage during request execution. Specifically, for each dataset and GNN model pair, we profile the peak memory usage in inference container for a small amount of requests, then we train a 2-layer MLP model with a hidden dimension of 16 to predict peak memory usage of each request during the inference stage. The input of the MLP model consists of the numbers of vertices and edges involved in each GNN layer's computation, which are determined by the neighbor subgraph of the query vertex. The trained model shows great capacity in peak memory prediction, with the difference between predicted and actual memory usage being less than 10%. Meanwhile, the computational overhead of the prediction model is negligible, with the prediction latency being less than 1ms. Note that the training process only requires minor effort and takes few seconds to complete using CPU, and we only re-train the model when employing datasets with distinct feature dimension or changing GNN models. Since prediction error could happen, for robust serving, we adopt a simple fault tolerant strategy, where we clear the cache and re-execute the request when the prediction error could potentially lead to OOM.

Cache eviction. Cache eviction occurs when detecting memory insufficiency for execution after memory usage prediction. For maintaining cache effectiveness, the feature cache adopts degree-based eviction strategy to decide which data should be preserved. In detail, during eviction, the vertices with small degree in origin graph will be prioritized to be evicted, since vertices with smaller degree are less likely to be sampled and reused. As for the number of entries to be evicted, feature cache first computes the size of memory needed to be reclaimed from cache based on predicted memory usage and container memory capacity. Then it evicts as few entries as possible to just meet the memory reclaim requirement. Additionally, features related to current neighbor subgraph will be preserved during eviction regardless of the degree.

4.4 Load-aware Request Scheduler

Serverless GNN inference suffers from long container cold startup latency, which can even exceed request processing time due to the complex environment required by GNN execution. Current serverless scaling policy [10] launches a new container if no idle container is available upon request arrival, and processes the request in the newly launched container after its initialization. When adopting such a policy for serverless GNN inference, the long cold startup latency incurs load imbalance. This is because while some requests are waiting for cold startup, other containers may finish processing requests and become idle. Such load imbalance indicates that the cold startup latency can be bypassed through dynamically scheduling requests to idle containers. Therefore, FaaSGNN runs a *load-aware request scheduler* to support dynamic scheduling, and each node independently schedules requests for execution with its local scheduler when deployed on multiple nodes.

Request queuing. The scheduler maintains a first-come-first-serve (FCFS) request queue for each container, and lets each container continue to process request at the front of its associated queue until the queue becomes empty. When a request arrives with no idle container available, the scheduler pushes the request into the queue of the warm container with minimum queue length. As the request can get processed by existing warm container after a short queuing time, the long cold startup latency is reduced effectively.

Determine container launch. As for container launching, current policy initializes a new container when request arrives with no available containers. However, such a policy can result in resource over-provisioning due to container's long cold startup latency. More specifically, during the period when some containers are undergoing cold startup, the supported concurrency of the system may be able to meet the requirement of incoming traffic after those containers finish initialization. Meanwhile, requests that trigger container launches can be processed through request queuing, which reduces number of pending requests. As a result, naively launching containers whenever there is no idle container leads to resource waste.

To determine when to launch new containers, the scheduler adopts a scaling metric which considers current system workload and number of containers to decide container launches. When a request arrives, a new container is launched only if the condition in Eq. 2 is met and no idle container is available:

$$n_{req} > n_{container}, \tag{2}$$

where n_{req} denotes the number of requests that are either waiting or under execution in the system, and $n_{container}$ denotes the number of containers, including both warm containers and containers undergoing cold startup. When n_{req} exceeds $n_{container}$, the scheduler considers the supported concurrency of the system insufficient, and launches a new container. Otherwise, the incoming request is scheduled to the request queue without triggering container launch.

Dynamic re-scheduling. During processing requests with request queuing, a container may become idle while some containers still have pending requests in their associated queues. Such phenomenon occurs either when a container finishes cold startup, or when a warm container finishes processing its request queue in advance due to the varying execution time of requests. To further balance workloads among containers, the scheduler dynamically re-schedules requests when it detects idle containers. Specifically, when a container becomes idle, the scheduler selects a request from the tail of a non-empty queue if it exists, then re-schedules this request to idle container for immediate execution. In the special case

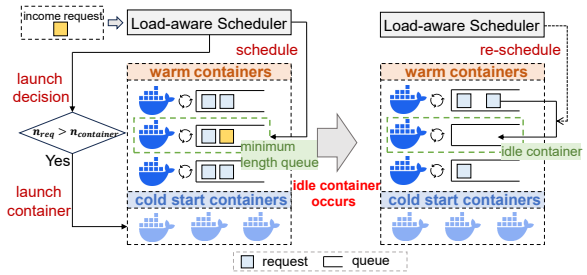


Figure 11: An example of load-aware request scheduler.

where a request arrives with no warm container, the incoming request will be placed in a temporary queue, and will be re-scheduled to idle container once any container finishes cold startup.

Figure 11 shows an example of the mechanism of load-aware scheduler. When receiving incoming request (the left part), the scheduler first determines whether to launch new container. If the number of unfinished requests in current system is larger than the number of containers, a container launch will be triggered. Then the scheduler tries to allocate an idle container to execute the incoming request. In case no idle container is available, the scheduler selects the warm container with minimum queue length and schedules the request to this container. When an idle container becomes available resulting from either finishing cold startup or having empty request queue (the right part), the scheduler selects a request from the tail of a non-empty queue and re-schedules the request to the idle container, thus balancing loads among containers.

4.5 Implementation

We implement FaaSGNN in Python and C++ code based on FaaSFlow [35], a serverless system designed for efficient serverless workflow execution. Specifically, we implement our *load-aware request scheduler* and integrate it with serverless system for request scheduling and container management, while *serverless-native on-demand graph fetching* strategy and *memory-aware adaptive feature caching* policy are implemented in GNN functions with system modified to support these features. We employ Docker [2] to create containers and use local HDD to simulate remote storage. We rely on HTTP to facilitate communications between hosts and containers, and all containers are set with a 10-minute keep-alive timeout.

When deployed on multiple nodes, FaaSGNN launches load-aware scheduler on each node for managing local containers and request scheduling. Besides, FaaSGNN specifies a head node to run the gateway, which receives incoming requests and dispatches requests to each node's scheduler in the round-robin manner. As each node independently manages its function execution, the scaling overhead primarily arises from the communication involved in dispatching requests across nodes, which remains minimal when cluster size increases.

5 Evaluation

This section evaluates FaaSGNN in improving average end-to-end latency and memory usage, followed by ablation studies. Furthermore, we analyze the performance of FaaSGNN under different load patterns and varying number of GNN layers, as well as the scalability and performance with GPU inference. Lastly, we discuss the monetary cost and the runtime overhead.

Table 1: Experimental setup

	Specification
Hardware	CPU: AMD EPYC 7302 @ 3.0GHz Cores: 64, DRAM: 125 GB, HDD Bandwidth: 500 MB/s
Software	OS: Ubuntu 20.04.5 LTS with kernel 5.15 Python: 3.9.19, GCC: 9.4.0, Docker: 20.10.18 PyTorch: 2.3.1, DGL: 2.1.0
Container	Runtime: Python 3.9.19, Linux with kernel 5.15, Lifetime: 10 min
Benchmark	GCN [28], GAT [55]

Table 2: Dataset statistics

Graph	# Vertices	# Edges
reddit-hyperlink (RE)	35,776	286,561
higgs-twitter (TW)	456,626	14,855,842
stackoverflow (ST)	2,601,977	63,497,050
ogb-paper (PA)	111,059,956	1,615,685,872

5.1 Experimental Setup

We evaluate FaaSGNN on a 2-node cluster with each node equipped with 64 cores and 125 GB memory. Table 1 describes the detailed hardware and software setup.

Benchmark: We use GCN [28] and GAT [55] that are representative GNN models and widely used by previous works [41, 54] to evaluate the performance of FaaSGNN. GCN is the backbone network for many other GNNs such as GraphSAGE [22], while GAT differs from GCN in aggregation operator. For GCN, we use 2 layers with 16 hidden dimension. For GAT, we use 2 layers with 8 attention heads, and each head has hidden dimension 8. We also evaluate FaaSGNN using the two models with 3 and 4 layers in Section 5.5.

Graph dataset and trace: We use four representative real-world datasets as listed in Table 2 for evaluation, including *reddit-hyperlink* (RE), *higgs-twitter* (TW) and *stackoverflow* (ST) that are three social networks [29] at different scales and one large citation network *ogb-paper* (PA) [8]. We generate random features with dimension 300, 300, 512 for RE, TW, ST respectively as features are not provided in original datasets. We conduct most experiments on these three datasets due to cluster resource limitation, and we evaluate FaaSGNN on PA dataset in Section 5.6. Microsoft Azure Function trace [50] is used to simulate the request arrival pattern in real-world serverless applications. We randomly select 2 traces from anonymous functions that are frequently invoked (total invocations larger than 10000) (denoted as "*trace-0*" and "*trace-1*") to conduct the experiments. Other frequent traces show similar result according to our measurement. As each graph dataset possesses an ID sequence recording the order of vertex access, we map the invocation timestamps in the trace to ID sequences of different datasets and generate requests with vertex from ID sequences. Due to the resource limitation of our cluster, we scale the trace to guarantee that the peak load in the trace can be handled by our cluster, without changing the pattern of invocation.

Baselines: We compare FaaSGNN with DGL-serverless and AWSGNN [72]. For both baseline systems, we profile the peak memory usage when processing query vertex with maximum in-degree

Figure 12: The average end-to-end latency (bars) and the normalized memory usage (lines) of DGL-serverless, AWSGNN and FaaSGNN.

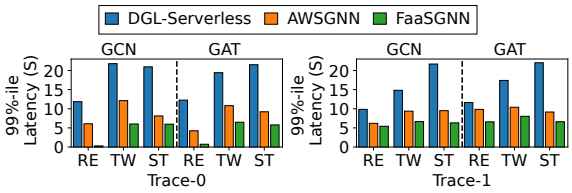


Figure 13: The 99%-ile latency of DGL-serverless, AWSGNN and FaaSGNN.

in different datasets, and configure the container memory capacity to the corresponding value when testing on different datasets, with the CPU quota in proportion to memory [6].

5.2 Reducing Latency and Memory Usage

Figure 12 shows the average end-to-end latency with DGL-serverless, AWSGNN and FaaSGNN. As observed, FaaSGNN speeds up at 5.65× and 1.25× on average end-to-end latency compared with DGL-serverless and AWSGNN, respectively.

FaaSGNN achieves the best latency performance by minimizing long data fetching latency and mitigating long cold startup latency. In contrast, DGL-serverless suffers from high data fetching overhead due to full graph and feature fetching in cold containers. Although AWSGNN's on-demand fetching reduces latency compared to DGL-serverless, it still faces inefficiencies due to fetching all required features for each request. Both two baselines also suffer from long cold startup latency.

FaaSGNN also achieves 57.1% and 8.1% lower memory usage on average compared to DGL-serverless and AWSGNN. DGL-serverless consumes the largest memory, because it fetches entire graph topology and features in each container. Meanwhile, during request spike, long data fetching latency in cold containers will lead to more container launches according to scaling policy.

Besides average latency, Figure 13 shows the 99%-ile latency of FaaSGNN and baselines. FaaSGNN reduces the 99%-ile latency by 7.27× and 3.51× compared to DGL-serverless and AWSGNN, respectively. In FaaSGNN, the adaptive caching policy reduces the data fetching latency, while load-aware scheduler further reduces latency caused by load imbalance under varying benchmarks.

5.3 Ablation Study

5.3.1 Effectiveness of On-demand Graph Fetching. To evaluate the effectiveness of on-demand fetching strategy, we compare the data fetching duration of FaaSGNN and DGL-Serverless which adopts full-data fetching scheme. Since DGL-Serverless only fetches data in

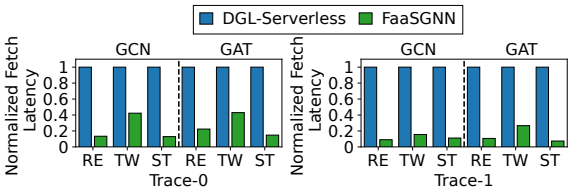


Figure 14: The normalized fetching duration of DGL-serverless and FaaSGNN.

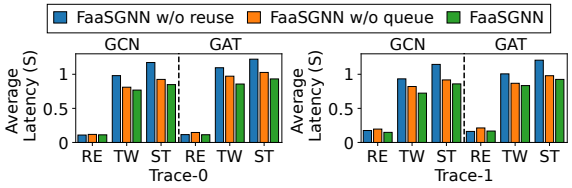


Figure 15: The end-to-end latency of FaaSGNN, FaaSGNN w/o reuse and FaaSGNN w/o queue.

cold containers, we only compare the fetching duration with regard to cold containers. Figure 14 shows the normalized data fetching duration of FaaSGNN and DGL-Serverless under different settings. As observed, FaaSGNN significantly reduces the feature fetching latency, with an average of 5.17× and 8.89× lower duration in the two traces compared to DGL-serverless. This is because FaaSGNN only requires fetching an average of 4.25% amount of vertices rather than the whole graph, compared with DGL-Serverless. Meanwhile, with less data fetched into the container, the memory capacity of the container in FaaSGNN can be set to a lower value to avoid memory waste.

5.3.2 Effectiveness of Feature Caching Policy. To evaluate the effectiveness of adaptive feature caching policy, we compare FaaSGNN with *FaaSGNN w/o reuse*, a variant that disables the feature cache. Different from FaaSGNN, *FaaSGNN w/o reuse* fetches all features associated with sampled neighbor subgraph from remote storage for every request. Figure 15 compares the performance of average end-to-end latency of the two systems under different settings. As observed, FaaSGNN achieves an average of 1.20× and 1.21× lower latency with trace-0 and trace-1 compared to *FaaSGNN w/o reuse*. This demonstrates that feature cache effectively reduces the data fetching latency in warm containers. In more detail, FaaSGNN can reuse an average of 86.1% (up to 94.4%) of vertices with associated features under different settings. The overlap rate of requests processed in FaaSGNN is calculated from dividing the number of reused vertices by number of vertices in sampled subgraph of each request. Overall, adaptive caching shows great performance in utilizing data overlap without additional memory consumption.

5.3.3 Effectiveness of Load-aware Scheduler. To evaluate the effectiveness of load-aware scheduler, we compare FaaSGNN with *FaaSGNN w/o queue*, a variant of FaaSGNN that disables the load-aware scheduler. For *FaaSGNN w/o queue*, the request is first scheduled to the idle warm containers. If there is no idle container, the scheduler launches a new container, and the request gets processed by this container after it finishes initialization Figure 15 presents the

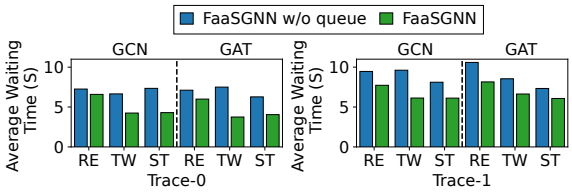


Figure 16: The average request waiting time in FaaSGNN and FaaSGNN w/o queue.

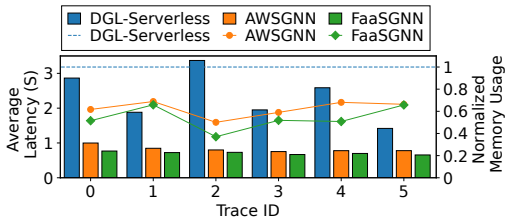


Figure 17: Average end-to-end latency (bars) and normalized memory usage (lines) of DGL-serverless, AWSGNN, and FaaSGNN on 6 different traces.

performance of FaaSGNN and *FaaSGNN w/o queue* under various settings.

As observed, *FaaSGNN w/o queue* increases the average latency by 1.12× for trace-0 and 1.14× for trace-1, respectively. We further study the waiting time of requests upon whose arrival there is no idle container available. Figure 16 shows the average waiting time of these requests before getting processed. As shown, *FaaSGNN w/o queue* experiences long waiting time because of the high initialization overhead for launching new containers. Overall, the results highlight the effectiveness of load-aware scheduler in mitigating high latency incurred by container cold start.

5.4 Performance with More Load Patterns

We evaluate FaaSGNN using more load traces that show different characteristics of load spikes. Without loss of generality, we use GCN model to test performance on the TW dataset, and other GNN-dataset combinations have similar results. As shown in Figure 17, FaaSGNN reduces the latency by 67.6% and 13.9% on average compared to DGL-serverless and AWSGNN. The lines show the memory usage normalized to DGL-serverless, with FaaSGNN consuming 46.1% and 14.2% less memory than DGL-serverless and AWSGNN, respectively. FaaSGNN is effective with varying load patterns.

5.5 Effectiveness for GNNs with More Layers

As an example, Figure 18 shows the average end-to-end latency and memory usage of different systems with two GNN models on TW dataset mapped with trace-0, when we increase the number of layers. More layers would lead to over-smoothing problem [47].

As observed, FaaSGNN consistently outperforms baseline systems across all settings, with the latency reduced by 50.7% and 12.5% on average compared to DGL-Serverless and AWSGNN respectively. For memory usage, FaaSGNN achieves a 46.2% and 27.0% lower usage on average, respectively. The results demonstrate the

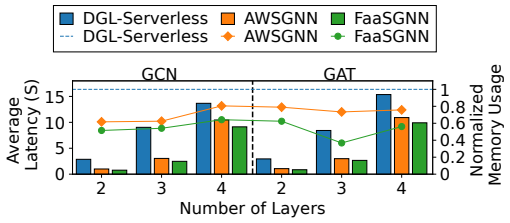


Figure 18: End-to-end latency (bars) and normalized memory usage (lines) of DGL-Serverless, AWSGNN, and FaaSGNN across GNN models with different layers.

Table 3: The average latency (in seconds) and memory usage (in 10^5GB·s) of different systems tested on PA dataset.

GNN Model	Metric	DGL-Serverless	AWSGNN	FaaSGNN
GCN	Latency	OOM	1.96	0.45
	Memory		9.76	2.15
GAT	Latency	OOM	4.05	0.48
	Memory		14.5	2.29

effectiveness of FaaSGNN across various number of GNN layers, highlighting its superiority in different GNN inference scenarios.

5.6 Scalability

In this subsection, we evaluate the scalability of FaaSGNN across two dimensions, i.e. scaling to large graph dataset and multiple physical nodes, for comprehensive evaluation.

(i) Scaling to large graph dataset. We evaluate the performance of DGL-Serverless, AWSGNN and FaaSGNN on PA dataset, a large real-world graph dataset with over 1 billion edges. As the vertex ID sequence that represents the order of vertex access is not specified in PA dataset, we construct the sequence from edges by concatenating two endpoints' ID of each edge following the order in edge list. Subsequently, we map the invocation timestamps in trace-0 to this sequence for request generation.

Table 3 shows the average end-to-end latency and memory usage of different systems on PA dataset. Note that DGL-Serverless runs out of memory (OOM) resulting from the following two reasons: (1) The latency of fetching entire PA dataset into memory is extremely high, which consequently triggers more container cold startups when no warm container is available; (2) Each container requires large memory for storing entire PA dataset, and OOM error occurs as the number of containers increases. For the remaining two systems, as observed, FaaSGNN consistently shows low average latency and memory usage compared with AWSGNN across various settings. In detail, FaaSGNN reduces latency by 77.0% and 88.1% with GCN and GAT respectively, while consumes 78.0% and 84.2% less memory than AWSGNN. Such results indicate the necessity of on-demand fetching when encountering large graphs, and demonstrate the effectiveness of FaaSGNN when the scale of graph grows.

(ii) Scaling to more nodes. This experiment evaluates FaaSGNN when the scale of cluster increases. We use instances of `ecs.g8i.8xlarge` from ECS [4], with each node equipped with 32 CPU cores and 128 GB memory. We adopt round-robin manner

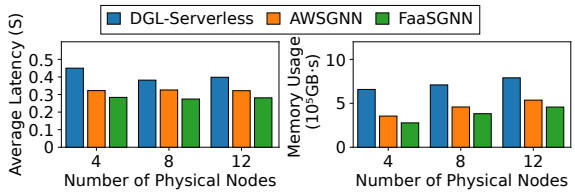


Figure 19: Average end-to-end latency (left) and memory usage (right) of DGL-Serverless, AWSGNN and FaaSGNN on clusters with various numbers of physical nodes.

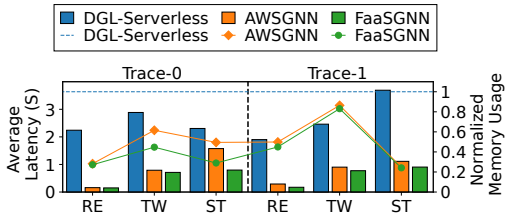


Figure 20: Average end-to-end latency (bars) and normalized memory usage (lines) of DGL-serverless, AWSGNN, and FaaSGNN with inference on GPU.

to dispatch requests across physical nodes, and apply the same load to clusters at different scales.

Figure 19 shows the average end-to-end latency and memory usage using GCN model on TW dataset across clusters at different scales. FaaSGNN reduces the end-to-end latency by 31.8% and 13.5%, and reduces the memory usage by 48.2% and 17.2% on average compared with DGL-Serverless and AWSGNN, respectively. Moreover, the latency of FaaSGNN remains almost the same when tested across clusters at different scales, indicating that FaaSGNN preserves strong scalability with minor scaling overhead.

5.7 Adapting to GPU

Without loss of generality, we use GCN model to evaluate the performance of FaaSGNN, when the GNN inference is performed on accelerators like GPU (Nvidia Ada-6000 GPU here).

Figure 20 shows the average end-to-end latency of different systems on all three graph datasets under the two traces. As observed, FaaSGNN achieves speedup at 4.41× and 1.38× on average end-to-end latency compared to DGL-serverless and AWSGNN, respectively. Although GPU accelerates the model inference procedure, the data fetching overhead and latency incurred by cold startup are still preserved by DGL-serverless and AWSGNN.

For memory usage, we only take CPU side memory usage into account, as GPU memory usage is the same across all systems. FaaSGNN uses 66.4% and 21.2% lower memory compared to DGL-serverless and AWSGNN. FaaSGNN still has the lowest memory usage since it reduces the data fetching latency and container cold startup latency. Such latency leads to more container launches when request spikes in the two baselines.

5.8 Monetary Cost

To validate whether serverless is capable to reduce the monetary cost of GNN service, we compare FaaSGNN against *Serverful-GNN*, a serverful GNN inference service. Serverful-GNN deploys the service on a long-term alive server with entire graph topology and feature data stored in memory. For incoming request, Serverful-GNN starts new threads for executing sample and inference, without launching containers. We extract one day invocation pattern of the function with most invocations in MAF traces, and map invocation timestamps to vertex ID sequence in TW dataset.

For Serverful-GNN, we select `c4.2xlarge` as the server instance from AWS EC2 [3] that "just meets" the requirement for supporting peak resource usage in the one day trace. According to pricing model of AWS EC2 [3], the monetary cost of Serverful-GNN is 9.55$ (0.398$ per hour). For FaaSGNN, according to pricing model of AWS lambda [9], the monetary cost of FaaSGNN is 4.91$ ($1.67\times10^{-5}$$ per GB·s). After replaying the trace, FaaSGNN reduces the monetary cost by 48.6% compared with Serverful-GNN.

5.9 Overheads

Request scheduling, adaptive caching, and communication between hosts and containers introduce extra overhead in FaaSGNN.

Scheduling overhead: When receiving requests, FaaSGNN allocates containers and schedules requests for execution. The average CPU and memory usage for scheduling requests is 0.05 core and 478MB (0.37% of host memory capacity) respectively. The latency introduced by rescheduling for load balance is less than 0.1ms on average. **Cache overhead:** The adaptive feature cache introduces overhead during cache eviction and update. The latency for cache operations accounts for 2.6% (up to 3.9%) of end-to-end latency on average across various experiment settings. Meanwhile, FaaSGNN uses a prediction model to estimate memory usage and determine the number of eviction. For each request, the prediction time is less than 1ms on average. **Communication overhead:** The communication overhead comes from sending requests and controlling information between hosts and containers. The average communication latency is 2ms in various settings. The overhead is acceptable.

6 Conclusion

This paper proposes FaaSGNN, a serverless framework for GNN inference services that comprises a serverless-native on-demand graph fetching strategy, a memory-aware adaptive feature caching policy, and a load-aware request scheduler. The fetching strategy enables effective on-demand fetching of the required neighbor subgraph of query vertex from remote storage. In each inference container, FaaSGNN maintains a cache for frequently used features, allowing reuse in subsequent requests to further minimize data fetching latency. The scheduler dynamically re-schedules requests from request queue to idle containers and determines when to launch a new container based on real-time load. Experimental results show that FaaSGNN reduces end-to-end latency by 5.6× and memory usage by 57.1% compared to state-of-the-art methods.

Acknowledgment

We thank our shepherd Prof. Pankaj Mehra for the great suggestions in improving this paper. This work is partially sponsored by National Natural Science Foundation of China (62232011, 62432015), and Provincal Key Research and Development Program of Hubei (No. 2023BAB065). Quan Chen is the corresponding author.

References

[1] [n. d.]. Azure Machine Learning, ML as a Service. https://azure.microsoft.com/en-us/products/machine-learning/.

[2] [n. d.]. Docker: Accelerated Container Application Development. https://www.docker.com/.

[3] [n. d.]. EC2 On-Demand Instance Pricing - Amazon Web Services. https://aws.amazon.com/ec2/pricing/on-demand/.

[4] [n. d.]. Elastic Compute Service (ECS): Elastic & Secure Cloud Servers - Alibaba Cloud. https://www.alibabacloud.com/product/ecs/.

[5] [n. d.]. KServe. https://kserve.github.io/website/latest/.

[6] [n. d.]. Lambda quotas - AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html.

[7] [n. d.]. Machine Learning Service - Amazon SageMaker - AWS. https://aws.amazon.com/sagemaker/.

[8] [n. d.]. Node Property Prediction | Open Graph Benchmark. https://ogb.stanford.edu/docs/nodeprop/#ogbn-papers100M.

[9] [n. d.]. Serverless Computing - AWS Lambda Pricing - Amazon Web Services. https://aws.amazon.com/lambda/pricing/.

[10] [n. d.]. Understanding Lambda Function Scaling - AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html.

[11] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. doi:10.1109/SC41405.2020.00073

[12] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2022. Optimizing inference serving on serverless platforms. *Proc. VLDB Endow.* 15, 10 (June 2022), 2071–2084. doi:10.14778/3547305.3547313

[13] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages.

[14] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations*.

[15] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*. PMLR, 942–950.

[16] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 199–216.

[17] Anjan Chowdhury, Sriram Srinivasan, Animesh Mukherjee, Sanjukta Bhowmick, and Kuntal Ghosh. 2023. Improving Node Classification Accuracy of GNN through Input and Output Intervention. *ACM Trans. Knowl. Discov. Data* 18, 1, Article 17 (sep 2023), 31 pages. doi:10.1145/3610535

[18] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481.

[19] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[20] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 135–153.

[21] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA). 386–400.

[22] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc.

[23] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: fast distribution with lazy docker containers. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'16)*. USENIX Association, USA, 181–195.

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

[25] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive Sampling Towards Fast Graph Representation Learning. In *Advances in Neural Information Processing Systems*, Vol. 31. Curran Associates, Inc.

[26] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. 152–166.

[27] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv:1902.03383 [cs.OS]

[28] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.

[29] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[30] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. 2020. DADI: Block-Level Image Service for Agile and Elastic Application Deployment. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 727–740. https://www.usenix.org/conference/atc20/presentation/li-huiba

[31] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA.

[32] Xiao Li, Li Sun, Mengjie Ling, and Yan Peng. 2023. A survey of graph neural network based recommendation in social networks. *Neurocomputing* 549 (2023), 126441. doi:10.1016/j.neucom.2023.126441

[33] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 69–84.

[34] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. 2022. The Serverless Computing Survey: A Technical Primer for Design Architecture. *ACM Comput. Surv.* 54, 10s, Article 220 (sep 2022), 34 pages.

[35] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. FaaSFlow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland). 782–796.

[36] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. 2021. FlashCube: Fast Provisioning of Serverless Functions with Streamlined Container Runtimes. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems* (Virtual Event, Germany) *(PLOS '21)*. Association for Computing Machinery, New York, NY, USA, 38–45. https://doi.org/10.1145/3477113.3487273

[37] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. 401–415. doi:10.1145/3419111.3421281

[38] Chengzhi Lu, Huanle Xu, Yudan Li, Wenyan Chen, Kejiang Ye, and Chengzhong Xu. 2024. SMIless: Serving DAG-based Inference with Dynamic Invocations under Serverless Computing *(SC '24)*. IEEE Press, Article 38, 17 pages. doi:10.1109/SC41406.2024.00044

[39] Mingxuan Lu, Zhichao Han, Susie Xi Rao, Zitao Zhang, Yang Zhao, Yinan Shan, Ramesh Raghunathan, Ce Zhang, and Jiawei Jiang. 2022. BRIGHT - Graph Neural Networks in Real-time Fraud Detection. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management* (Atlanta, GA, USA) *(CIKM '22)*. 3342–3351. doi:10.1145/3511808.3557136

[40] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 443–458.

[41] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 443–458.

[42] Sudipta Mondal, Susmita Dey Manasi, Kishor Kunal, Ramprasath S, and Sachin S. Sapatnekar. 2022. GNNIE: GNN inference engine with load-balancing and graph-specific caching. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. 565–570. doi:10.1145/3489517.3530503

[43] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: an opportunistic caching system for FaaS platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 228–244.

[44] Jinyoung Park, Seongjun Yun, Hyeonjin Park, Jaewoo Kang, Jisu Jeong, Kyung-Min Kim, Jung-Woo Ha, and Hyunwoo J. Kim. 2025. Deformable Graph Transformer. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 47, 7 (2025), 5385–5396. doi:10.1109/TPAMI.2025.3550281

[45] Xiaoru Qu, Zhao Li, Jialin Wang, Zhipeng Zhang, Pengcheng Zou, Junxiao Jiang, Jiaming Huang, Rong Xiao, Ji Zhang, and Jun Gao. 2020. Category-aware Graph Neural Networks for Improving E-commerce Review Helpfulness Prediction. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management* (Virtual Event, Ireland) *(CIKM '20)*. Association for Computing Machinery, New York, NY, USA, 2693–2700. doi:10.1145/3340531.3412691

[46] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 753–767.

[47] T. Konstantin Rusch, Michael M. Bronstein, and Siddhartha Mishra. 2023. A Survey on Oversmoothing in Graph Neural Networks. arXiv:2303.10993 [cs.LG] https://arxiv.org/abs/2303.10993

[48] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. 2023. FlowGNN: A Dataflow Architecture for Real-Time Workload-Agnostic Graph Neural Network Inference. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1099–1112. doi:10.1109/HPCA56546.2023.10071015

[49] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Comput. Surv.* 54, 11s, Article 239 (nov 2022), 32 pages.

[50] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218.

[51] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 1–13.

[52] Xuehai Tang, Peng Wang, Qiuyang Liu, Wang Wang, and Jizhong Han. 2019. Nanily: A QoS-Aware Scheduling for DNN Inference Workload in Clouds. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2395–2402. doi:10.1109/HPCC/SmartCity/DSS.2019.00334

[53] Kiran K. Thekumparampil, Sewoong Oh, Chong Wang, and Li-Jia Li. 2018. Attention-based Graph Neural Network for Semi-supervised Learning. In *International Conference on Learning Representations*.

[54] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 495–514.

[55] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.

[56] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 267–281. https://www.usenix.org/conference/fast20/presentation/wang-ao

[57] Daixin Wang, Jianbin Lin, Peng Cui, Quanhui Jia, Zhen Wang, Yanming Fang, Quan Yu, Jun Zhou, Shuang Yang, and Yuan Qi. 2019. A Semi-Supervised Graph Attentive Network for Financial Fraud Detection. In *2019 IEEE International Conference on Data Mining (ICDM)*. 598–607. doi:10.1109/ICDM.2019.00070

[58] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[59] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 515–531.

[60] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. 2023. MGG: Accelerating Graph Neural Networks with Fine-Grained Intra-Kernel Communication-Computation Pipelining on Multi-GPU Platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, 779–795.

[61] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. 2023. TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, 149–164.

[62] Shu Wu, Yuyuan Tang, Yanqiao Zhu, Liang Wang, Xing Xie, and Tieniu Tan. 2019. Session-based recommendation with graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 346–353.

[63] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*.

[64] Guipeng Xv, Chen Lin, Wanxian Guan, Jinping Gou, Xubin Li, Hongbo Deng, Jian Xu, and Bo Zheng. 2023. E-commerce Search via Content Collaborative Graph Neural Network. 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, 2885–2897. doi:10.1145/3580305.3599320

[65] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. 417–434. doi:10.1145/3492321.3519557

[66] Liangwei Yang, Zhiwei Liu, Yingtong Dou, Jing Ma, and Philip S. Yu. 2021. ConsisRec: Enhancing GNN for Social Recommendation via Consistent Neighbor Aggregation. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Virtual Event, Canada) *(SIGIR '21)*. Association for Computing Machinery, New York, NY, USA, 2141–2145. doi:10.1145/3404835.3463028

[67] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: a native serverless system for low-latency, high-throughput inference *(ASPLOS '22)*. 768–781. doi:10.1145/3503222.3507709

[68] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 335–350.

[69] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J. Kim. 2019. Graph transformer networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*.

[70] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *International Conference on Learning Representations*.

[71] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1049–1062.

[72] Jian Zhang, Haozhu Wang, and Mengxin Zhu. 2022. Build a GNN-based real-time fraud detection solution using Amazon SageMaker, Amazon Neptune, and the Deep Graph Library. https://aws.amazon.com/blogs/machine-learning/build-a-gnn-based-real-time-fraud-detection-solution-using-amazon-sagemaker-amazon-neptune-and-the-deep-graph-library.

[73] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S. Warke, and Dean Hildebrand. 2018. Wharf: Sharing Docker Images in a Distributed File System. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 174–185.

[74] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.