ACM Symposium on Cloud Computing

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

Yuzhuo Yang[1], Kaihua Fu[1], Quan Chen[1], Deze Zeng[2], Shuo Quan[3], Jie Wu[4], Minyi Guo[1]

[1]Shanghai Jiao Tong University

[2]China University of Geosciences

[3]Cloud Computing Research Institute, China Telecom

[4]Temple University

SHANGHAI JIAO TONG UNIVERSITY

CHINA UNIVERSITY OF GEOSCIENCES

中国电信云计算研究院
China Telecom Cloud Computing Research Institute

Temple University

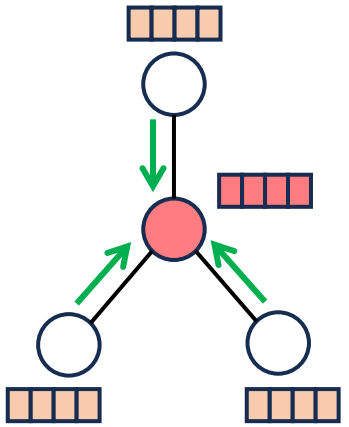
GNN inference prevails in cloud applications

- **Graph Neural Networks (GNNs)** are powerful in modeling graph structured data

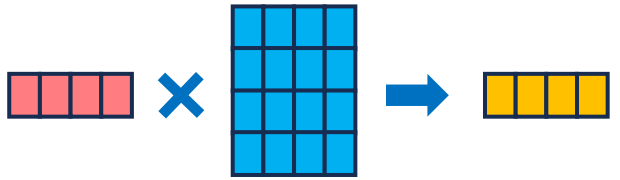
- GNN involved cloud applications:
  - E-commerce
  - Social Networks
  - Recommendation
  - …

- GNN computation
  - Two main operators: “aggregate” and “update”

Aggregate



Update



Serving GNN inference with serverless

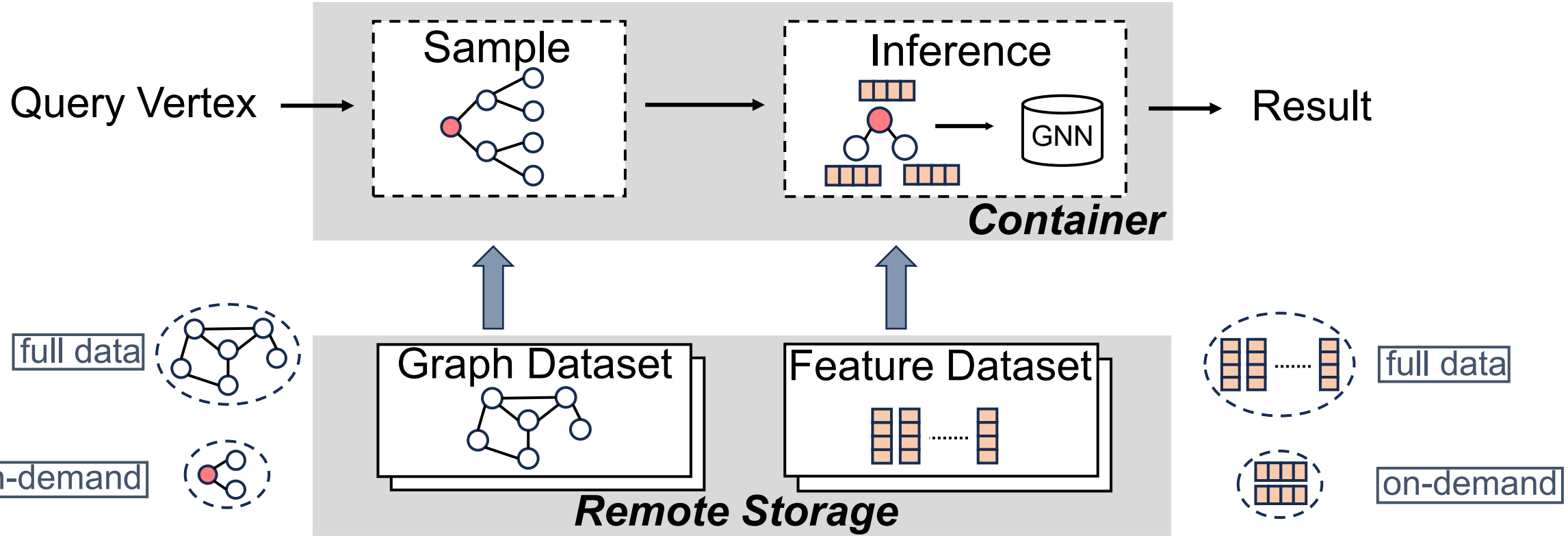
- **Serverless computing** is widely adopted by cloud venders

- Serverless computing efficiently handles load spikes for cloud applications:
  - Auto-scaling
  - Pay-as-you-go pricing
  - Ease of management
  - …

Cloud venders supporting serverless:

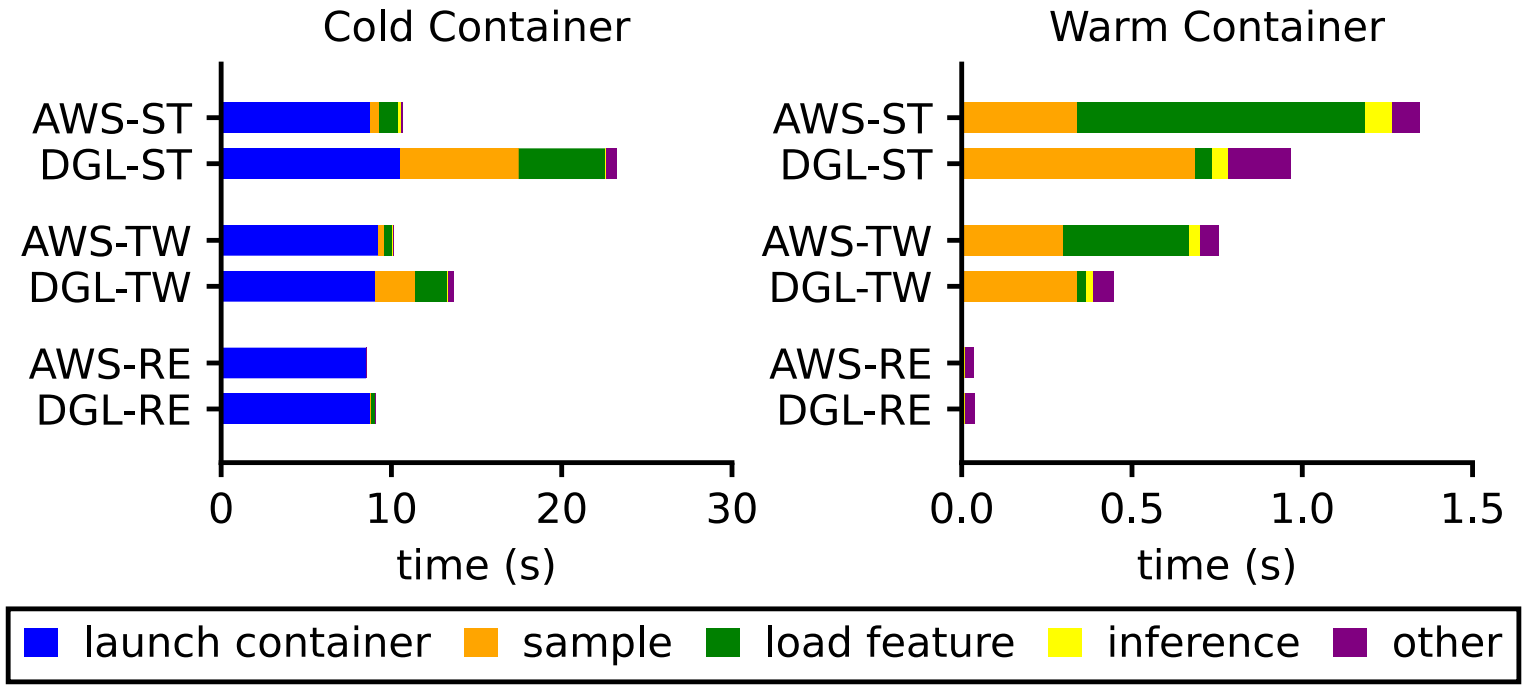
Workflow of serverless GNN inference

- Two computation phases: **Sample** and **Inference**
- Two data fetching schemes { Full-data fetching (e.g. *DGL-Serverless*); On-demand fetching (e.g. *AWSGNN*)



Data fetching & cold start latency

- **Latency Breakdown**

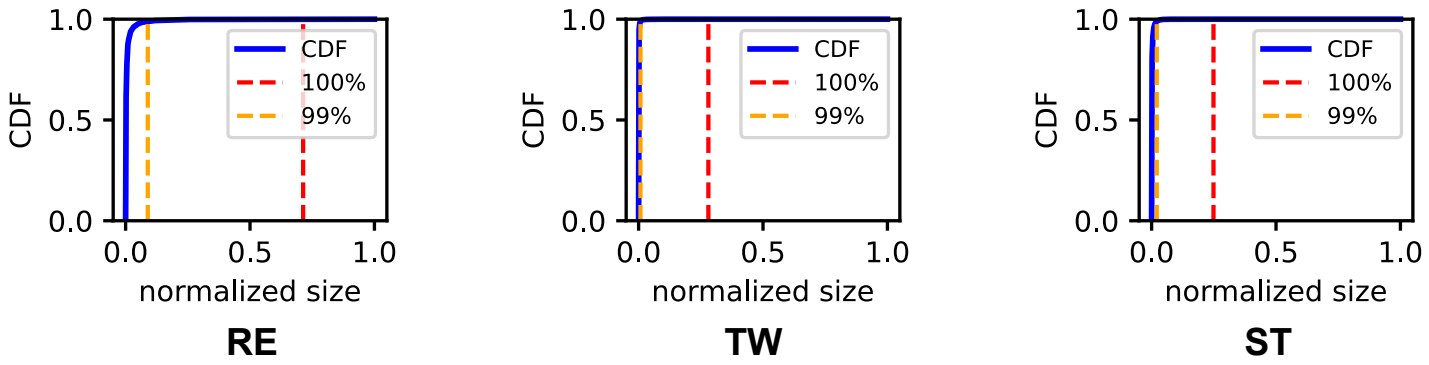


Large data fetching latency & cold start latency

Inefficiency of full-data fetching scheme

- **Large data fetching latency** in cold container
  - loading entire graph topology and feature data is time consuming …
- **Large memory usage**
  - large space required for storing data per container
  - substantial memory allocation during load spike

Our observation: only **a small proportion of data** is accessed for most requests with prevalent GNNs (e.g. 2-layer GCN).



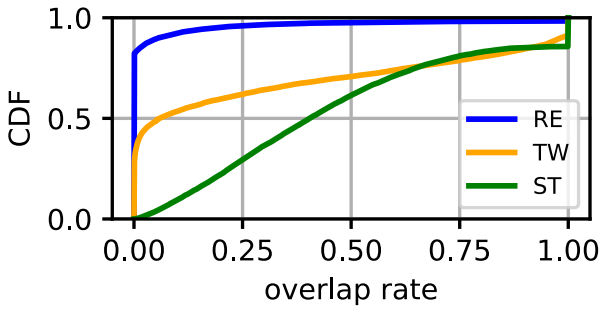
CDF of neighborhood size (normalized to full graph)

On-demand is promising

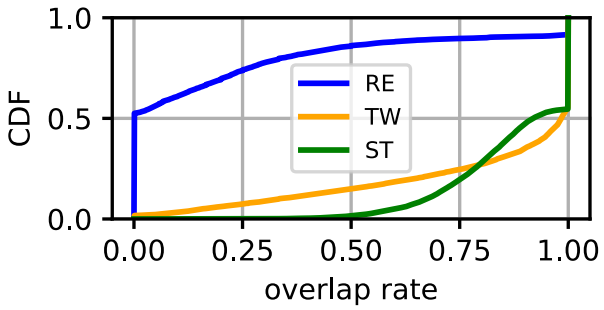
On-demand fetching is good, but…

- **Poor cost-efficiency** of implementation
  - employment of long-term graph database service cannot support fast scaling (*AWSGNN*)
- **Introduce fetching latency** in warm container
  - required neighborhood differs across requests

Our observation: there are **data overlaps** in neighbor subgraph across requests.



with preceding 1 request



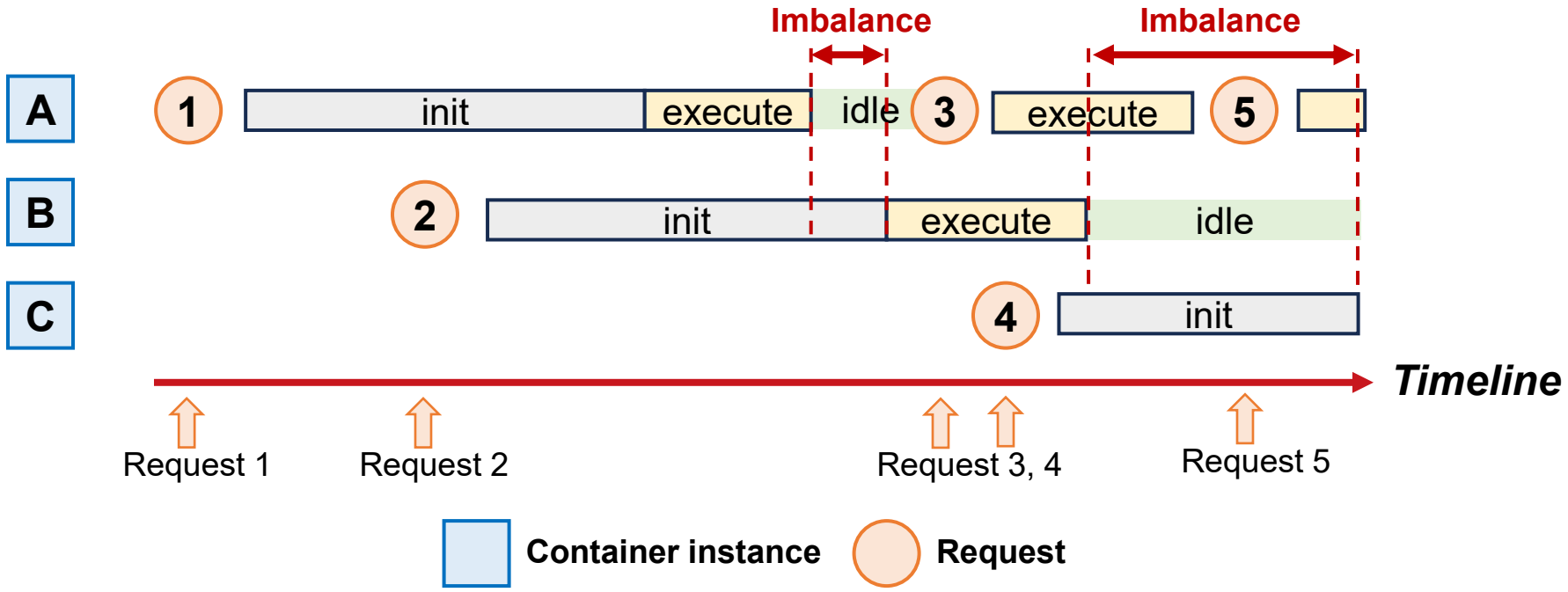
with preceding 10 request

CDF of data overlap rate (trace analysis)

Opportunity to reduce latency

Cold start incurred load imbalance

- Reflection on commonly practiced **scaling policy**
  - request undergoes cold start if no container is available upon arrival
- **Large cold start overhead** for GNN inference
  - heavy runtime initialization required by modern GNN framework (e.g. DGL)
  - exceeding request execution time



Design Objectives

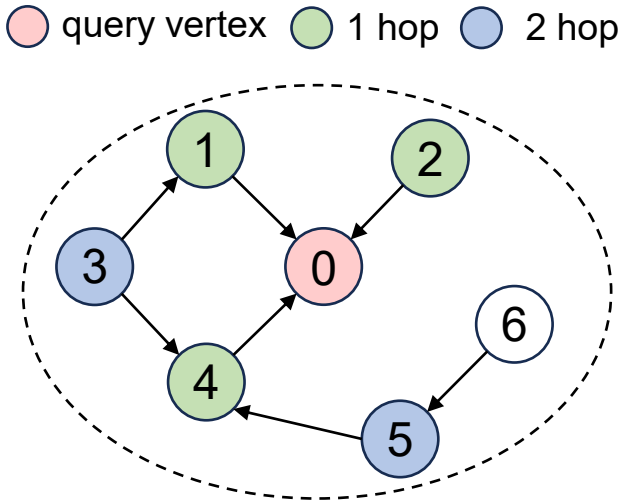
- **On-demand data fetching**
  - cost-efficiency
  - lightweight and fast subgraph extraction

- **Enabling data reuse**
  - effectiveness
  - reducing extra memory overhead

- **Mitigating cold start overhead**
  - load-balanced scheduling

Serverless-native on-demand graph fetching

- Fast subgraph traversal: **Compressed Sparse Row (CSR)**
  - two indices: **row index** and **column index**
  - subgraph extraction with iterative two-level index lookup



Remote Storage

row index:	0	3	4	4	4	6	7	7
column index:	1	2	4	3	3	5	6	

First round fetch

- Init frontier: [0]
- Row index lookup for vertex 0

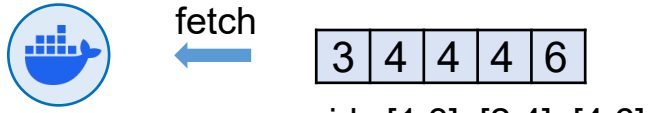
- Column index lookup for vertex 0

- Update frontier: [1, 2, 4]

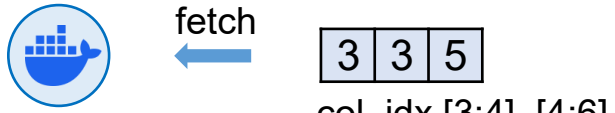
Finish hop-1 fetch

Second round fetch

- Init frontier: [1,2,4]
- Row index lookup for vertex 1,2,4



- Column index lookup for vertex 0



- Update frontier: [3, 5]

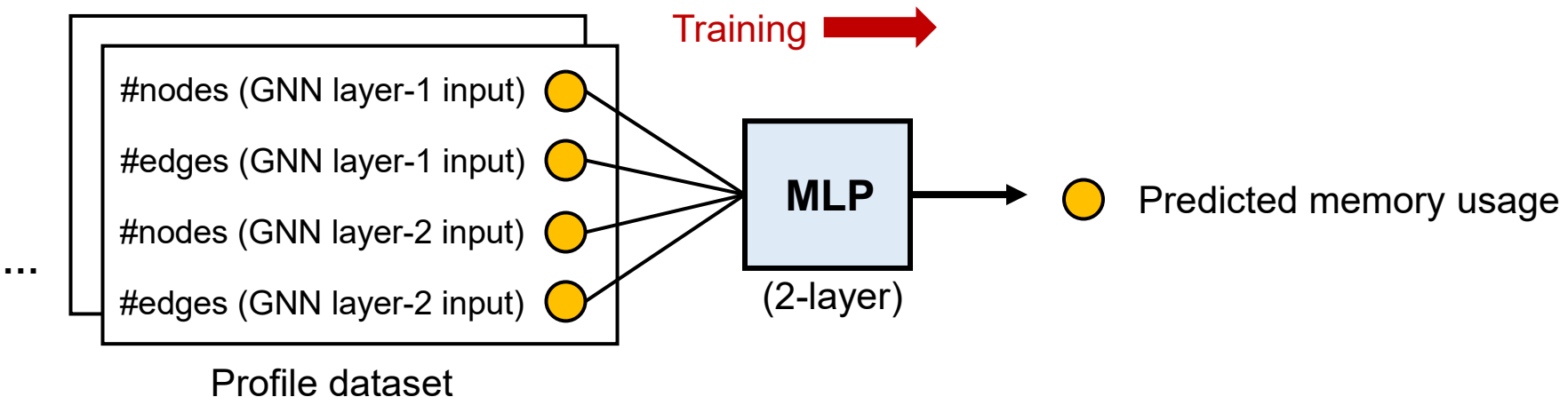
Finish hop-2 fetch

Efficient data reuse with dynamic caching

- Possible to achieve **caching within container memory**
  - memory usage diverse significantly across requests

- How to determine cache size?
  - static cache size loses effectiveness
  - dynamic caching without interference with execution

Our insight: memory usage can be accurately estimated after sampling.



- **Lightweight training** (within few minutes)
- **Minor inference overhead** (less than 1ms)
- **High accuracy**

Cache management

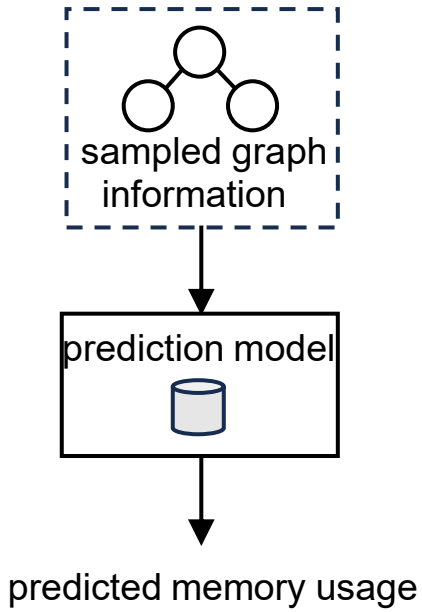
- What to cache? → feature data
  - accurate memory usage prediction after sampling
  - typically dominating fetching latency

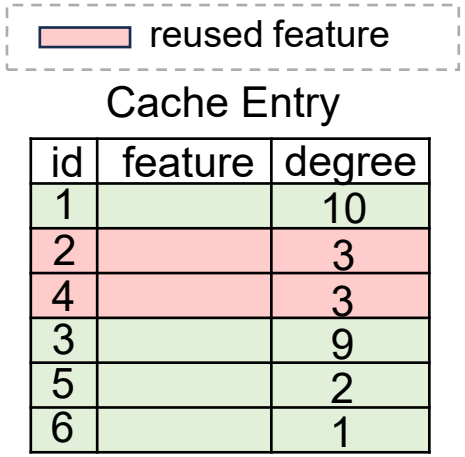
- Principles of **maximizing cache effectiveness**
  (i) continuously caching features of requests
  (ii) evict entries as less as possible when detecting memory insufficiency
  (iii) degree-based eviction

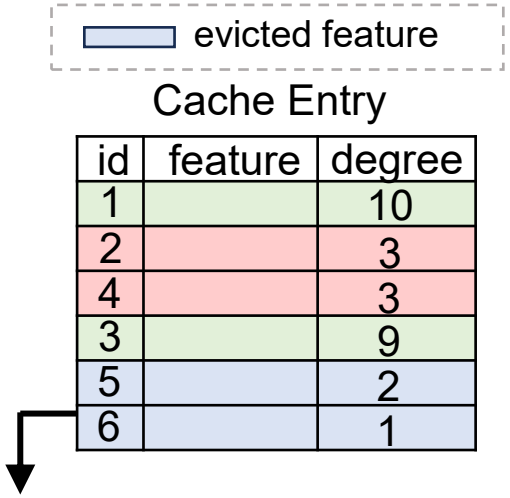
Vertex with **small degree** is prioritized to be evicted

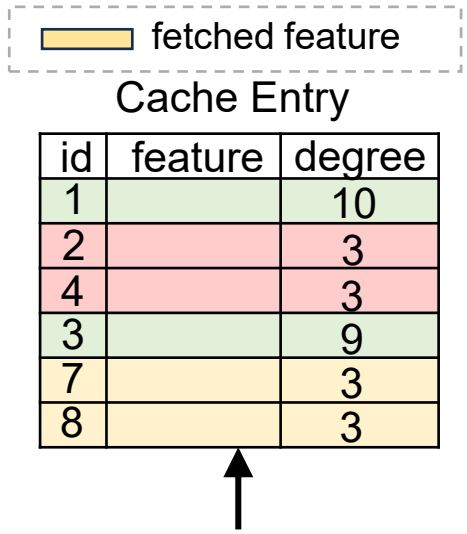
Cache workflow

- **Example**





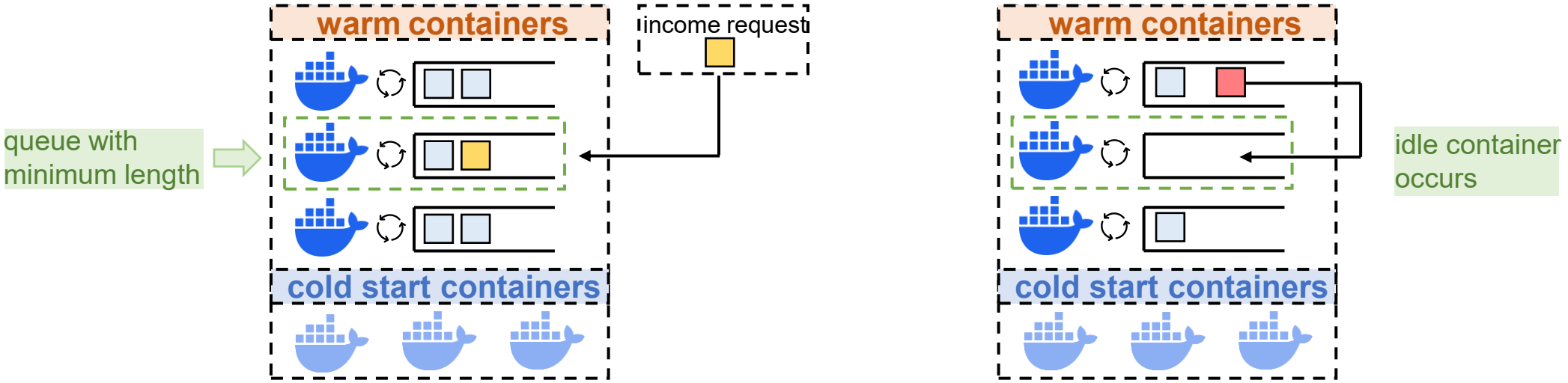




- Step 1: estimate memory usage
- Step 2: identify reused features
- Step 3: evict cache entries if memory insufficiency is detected
- Step 4: fetch un-cached data from remote, and collect features for inference

Scheduling to mitigate cold start overhead

- Dynamic scheduling to achieve **load balance**
  - enabling queuing for warm containers
  - dynamic re-scheduling to idle containers



- When to scale?
  - load of requests exceeds system capacity: $n_{req} > n_{container}$

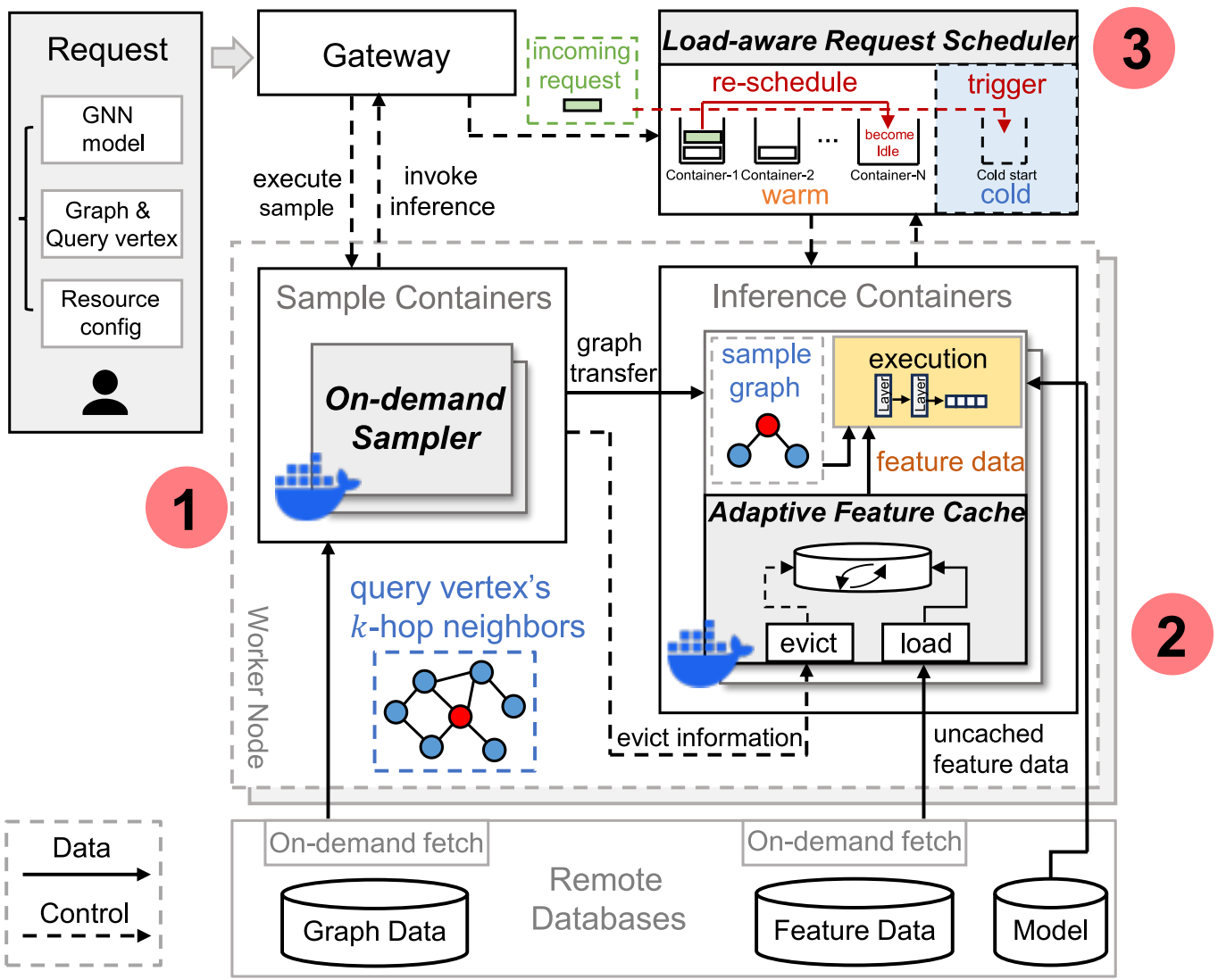
Put them together: FaaSGNN

- **Overview**

1. Serverless-native
   on-demand fetching strategy

2. Adaptive feature caching

3. Load-aware request scheduler



Evaluation Setup

- **Hardware & software configuration**

  - for each node:

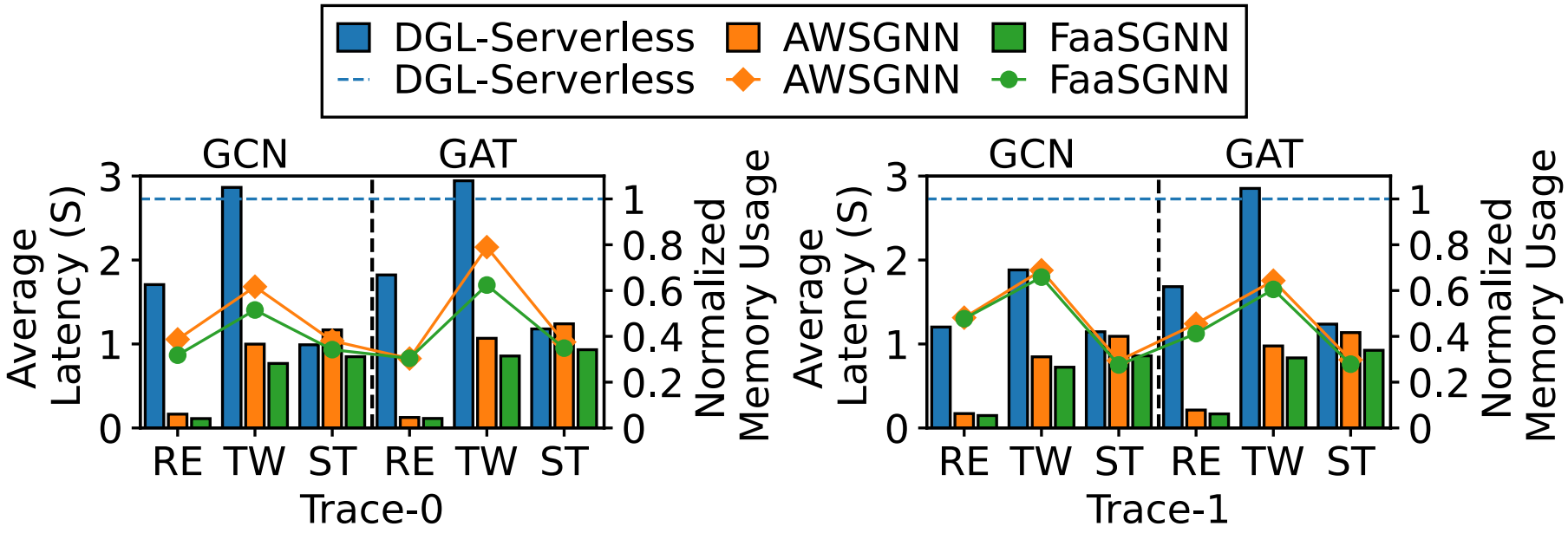
	Configuration
Hardware	CPU: AMD EPYC 7302 @3.0GHz Cores: 64, DRAM: 125GB, HDD Bandwidth: 500MB/s
Software	OS: Ubuntu 20.04.5 LTS, GCC: 9.4.0, Docker: 20.10.18 Python: 3.9.19, PyTorch: 2.3.1, DGL: 2.1.0

  - Cluster: 2 nodes (* scaling to 12 nodes in scalability evaluation with ECS instances)

- **Graph datasets:** Reddit-hyperlink (RE), Higgs-twitter (TW), Stackoverflow (ST) - [millions of edges]
  OGB-Paper (PA) - [billions of edges]
- **Trace dataset:** Microsoft Azure Function traces
- **GNN models:** GCN, GAT
- **Baselines:** DGL-Serverless, AWSGNN

Overall Performance: Latency & Memory Usage

- **Average end-to-end latency (bars) and normalized memory usage (lines)**

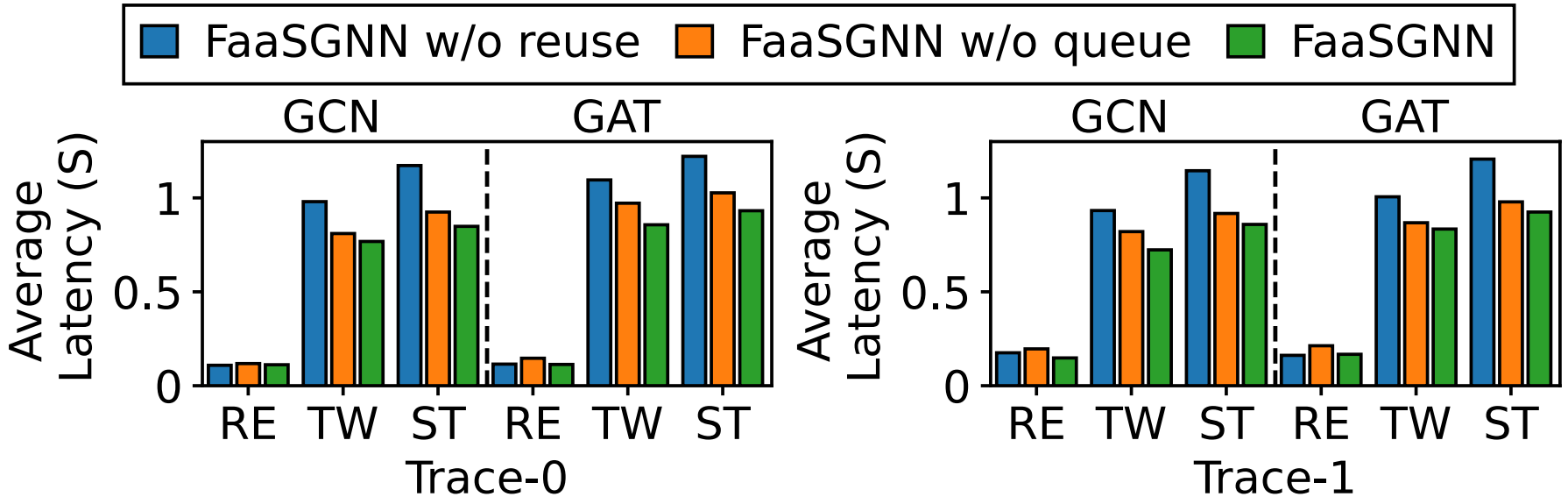


FaaSGNN consistently shows lower latency and memory usage under various settings compared to DGL-Serverless and AWSGNN

Ablation Study

- **Two variants**
  - *FaaSGNN w/o reuse*: feature cache is disabled
  - *FaaSGNN w/o queue*: dynamic scheduling is disabled

- **Average end-to-end latency:**

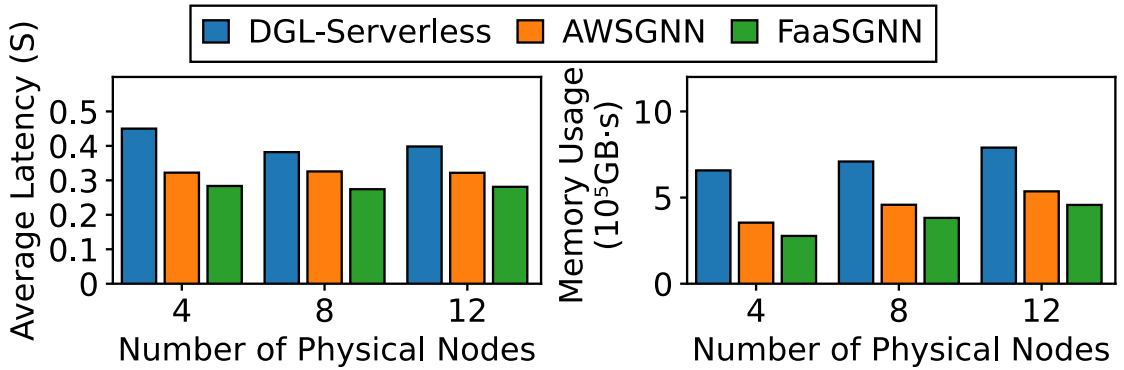


Scalability

- **Scale to large graph**
  - evaluation on PA dataset, which contains 1 billion edges

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

- **Scale to multiple physical nodes**
  - evaluation on clusters with 4, 8, 12 nodes



Conclusion

FaaSGNN: a serverless framework for GNN inference with low latency & memory usage

- **Observation**: large data fetching and cold start latency

- **Three key components**:
  - Serverless-native on-demand data fetching
    enable fast and cost efficient on-demand fetching with CSR-indices
  - Adaptive feature caching policy
    effectively leverage data overlap to reduce fetching latency
  - Load-aware request scheduler
    achieve load balance with dynamic scheduling

Thanks & QA

Contact: yang-yz@sjtu.edu.cn