INFless: A Native Serverless System for Low-Latency, High-Throughput Inference

Y. Yang, H. Zhang, L. Zhao, J. Li, Y. Li, M. Zhao (Tianjin University, TANKLAB), X. Chen (58.com),
K. Li (CIC, Tianjin University, TANKLAB)
ASPLOS 2022 Feb

Presenter: Mausam Basnet, University of North Texas

Date: 2024-12-05

Introduction

Serverless Platforms is widely adopted:

- Inference service easily separate from front-end applications
- Function template : quickly deployment without resource management
- There is auto scaling ability for varied workloads
- Cost saving because of pay-per-use billing model
- Eg: web services, IoT monitoring applications, entertainment

ML inferences : Deploying well-trained models in applications to provide prediction or classification services

- Unlike scheduled background applications:
    - ML inferences are often integrated into online websites
    - E.g., e-commerce, search engine, social network
    - Comes with strict latency requirements.

Background

China's largest local life service website relies on hundreds of ML inference services in their core business

- Deployed 600+ ML inference models
- Advertisements, and question-answering robots, to fraud detection
- Serving millions of requests every minute
- Latency critical with complex computation
- More than 90% of them need to respond within 200ms
- 400-server hybrid CPU/GPU cluster for inference

Inference Latency	Fraction of Models (%)
<50ms	86.2
50-200ms	11.6
200-500ms	1.1
500-1000ms	0.6
>1000ms	0.3

Table: Real-world latency SLO distribution by the local life service website.

Current Gaps

- Amazon Lambda, Google Cloud Functions, and Azure Functions do not provide GPU
- Do not provide latency guarantee
- High resource usage

Motivation

High Latency

The commercial serverless platform lacks the support of accelerators and therefore cannot provide low latency services for large-sized inference models.

No Accelerator Support: Platforms like AWS Lambda lack GPUs, leading to high latency for large models.

CPU-Memory Limit: CPU power is tied to memory size, making large models inefficient.

Latency Issue: Large models exceed 200ms even with maximum memory, failing to meet latency targets.

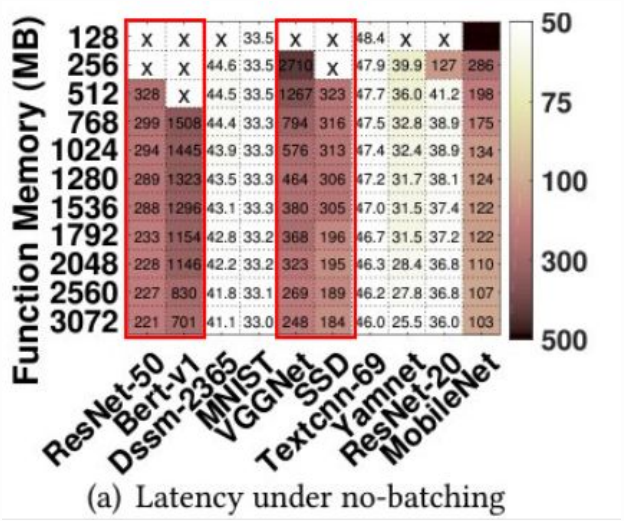


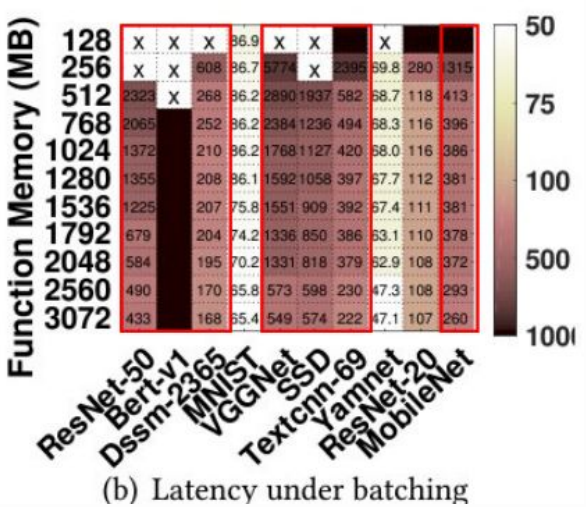
Fig: The inference latency distribution when running models on AWS Lambda without batching support

Motivation

No improvement for batch-enabled

For batch-enabled inference, commercial serverless platforms cannot provide low-latency services for some small-sized models.

- Small models face high latency even with batching on platforms like AWS Lambda.
- Batching increases execution time by 4x for models like DSSM-2365 and MobileNet.
- Even with batching, these models cannot meet low-latency requirements.



(b) Latency under batching

Fig: The inference latency distribution when running models on AWS Lambda with OTP-batching support

Motivation

Poor resource utilization

The one-to-one mapping request processing policy of commercial serverless platforms causes low resource utilization.

- Commercial platforms use a one-to-one mapping of requests to instances, causing inefficiency.
- This leads to excessive instances being created, especially under bursty workloads.
- This policy results in poor resource utilization and increased costs.

Total number of function invocations declines by 72%

Total number of launched instances under batching also declines by 35%

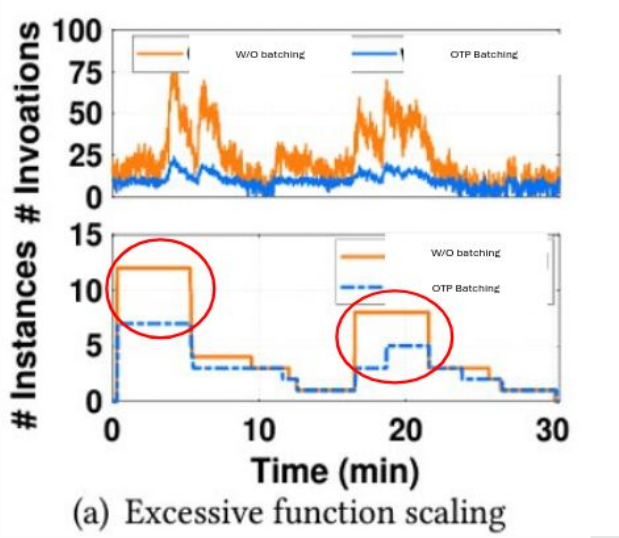


Fig: Excessive instances created by the one-to-one mapping policy (ResNet-20)

Motivation

Resource over-provisioning

- CPU is allocated based on memory size, leading to over-provisioning.
- More than 50% of memory is wasted to meet latency requirements for models like SSD.

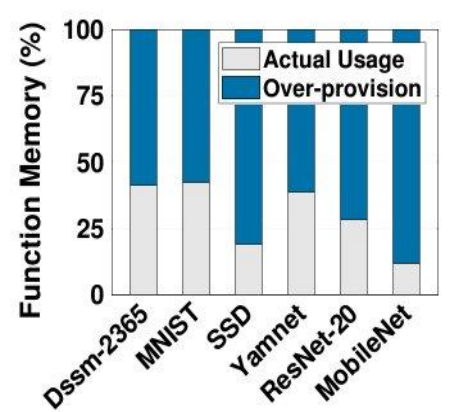


Fig: The memory over-provisioning for achieving low latency requirement

OTP batching limitations

- OTP batching adds another new buffer layer on top of existing system for batching
- It lacks control over scheduling delays and queuing inside serverless platforms.
- OTP batching uses a uniform scaling policy.

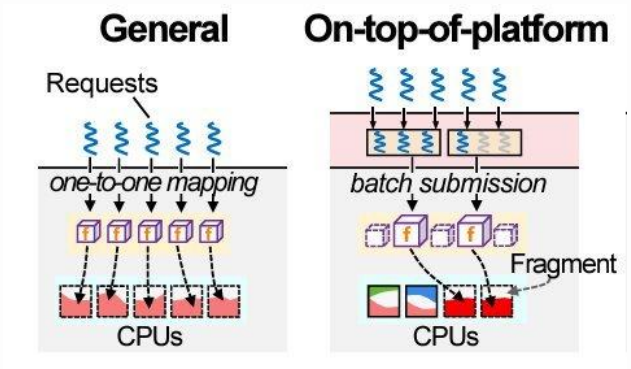


Fig: Schematic overview of serverless inference systems.

What should be done ?

High Latency → Support hybrid CPU/GPU

Resource over-provision → Function Profiling & Resource efficient scheduling

Low resource utilization → Function Profiling & Resource efficient scheduling

Inefficient OTP design → Supporting built-in batching

INFless System architecture

Exploit the features and characteristics of inference. e.g., shared operators, batching and computation intensive)

- Deployed by cloud providers
- Developers upload inference models
- Users can get SLO guaranteed inference service

Three major component/concepts:

- Function Profiling
- Built-in, Non-uniform batching
- Cold-start manager

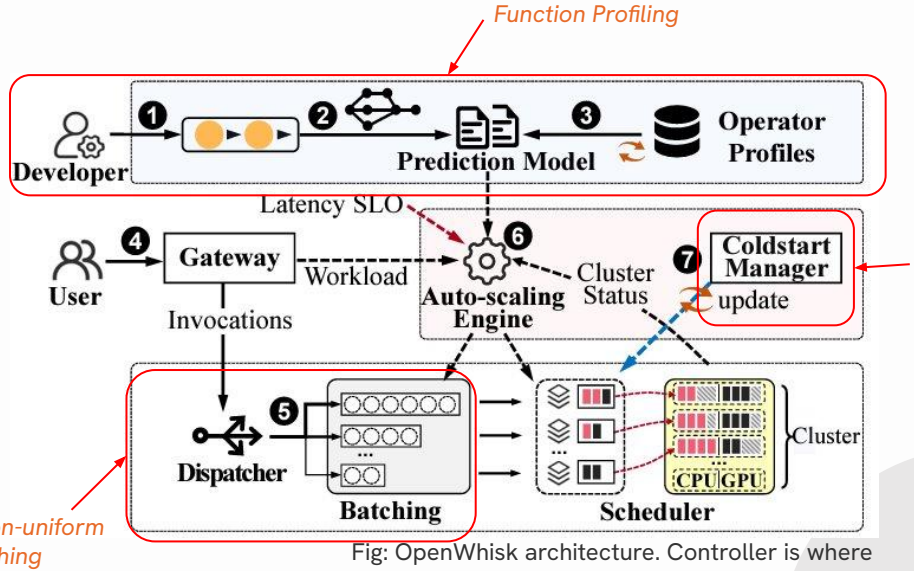


Fig: OpenWhisk architecture. Controller is where scheduling happens.

Built-in, Non-uniform Batching

Built-in: Batching is integrated into the serverless platform

- Has simultaneous & collaborative control over batchsize, resource allocation and placements

Non-uniform: Each instance has its own batch queue & have varying batch sizes and resource configurations

1. The request arrival rate for each instance is $[r_{low}, r_{up}]$

$$r_{up} = \lfloor \frac{1}{t_{exec}} \rfloor \times b, \quad r_{low} = \lceil \frac{1}{t_{slo} - t_{exec}} \rceil \times b$$

Suppose there are n instances of an inference function, each with their own r_{low} & r_{up}

- R_{max} = sum of r_{up} & R_{min} = sum of r_{low}

AutoScaling (Let R be the actual request per second for a function currently)

- If $R > R_{max}$, dispatch each instance with their r_{up} & launch new instances for processing $R - R_{max}$
- Each instance is allowed to have a varied RPS. Request arrival rate of an instance to be as close to its upper bound as possible to improve throughput
- In other case, when the requests are less, the auto-scaling engine will release extra instances

Combined Operator Profiling

Design a prediction model to estimate the latency under various batch sizes and resource configurations

- Inference models share common operators
- Execution time is dominated by a small subset of these operators.
- Instead of profiling every function, INFless profiles individual operators
- Estimates the execution time of entire models by aggregating the execution times of key operators
- Collected more than 100 operators' profiles and stored them in an operator profile database

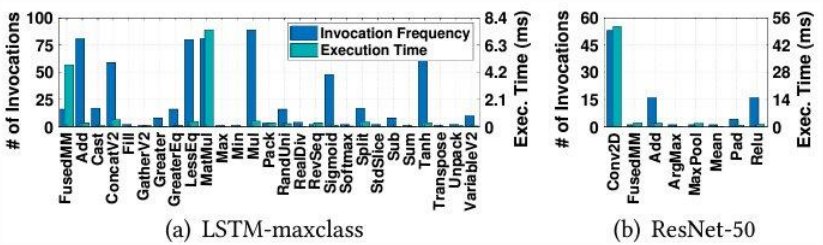


Fig: Calling frequency and execution time of the DNN operator

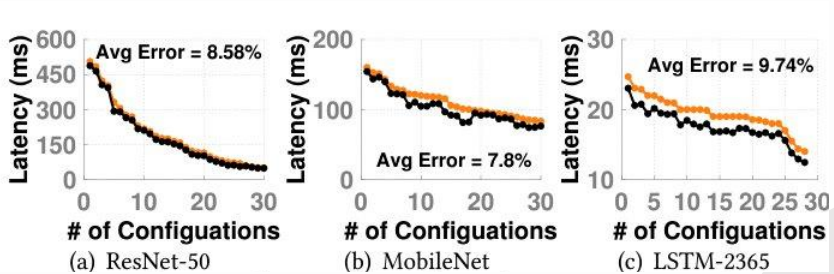


Fig: The prediction results of operator combination model across different batch-resource configurations.

Scheduling

Auto-scaling engine's functionality:

- Continuously tracks the incoming request rate (RPS) to assess if current instances can handle the load.
- Routes part of the requests to existing instances, reducing delays.
- Calls a scheduling algorithm to launch new instances if needed to handle extra requests.
- Uses an inference performance model to find the optimal instance configuration

Scheduling Objectives

- Minimize overall resource (CPU & GPU) usage on each server.
    - Ensures instance execution time meets the SLO latency. [$t_{wait} + t_{exec} \leq t_{slo}$]
    - Total resources used by instances on a server don't exceed the available resources of that server.
    - Ensure that the arrived requests R_k toward function k can be fully processed by all of its launched instances
    - Ensures batch sizes and configurations stay within allowed limits.

This optimization problem is at least as hard as the known NP hard bin packing problem

Scheduling Algorithm

Efficiently allocate instances to handle residual RPS while meeting latency SLOs and optimizing resource use.

Input: residual RPS of function k, batchsize set of k, available resources of m-server cluster, latency SLO of function k

Output: The number of instances for function k, resource config. for each new instances, placement of those instances

1. Repeat until residual RPS**, $R_k > 0$:**

2. Batch Size Loop: Iterate over batch sizes (starting with the largest)

3. Find Configurations (AvailableConfig Function):

- For each batch size, identify CPU/GPU configurations that meet the latency SLO.
- Predict execution time for each configuration and filter to find viable options.

4. Calculate Resource Efficiency:

- For each valid configuration, compute resource efficiency.
- Choose the configuration with the highest efficiency to optimize resource use.

5. Instance Allocation:

- Assign an instance with the selected configuration.
- Update remaining RPS and repeat until all RPS is processed.

Managing Cold Starts with LSTH

Cold starts significantly degrade performance, especially for inference functions with large models.

Traditionally used: Hybrid Histogram Policy (HHP)

- Tracks the idle times of a configurable duration (e.g., 4 hours)
- Derives two parameters **pre-warming window** and **keep-alive window**

This method is found to be too conservative and generates too much resource waste.

Long-Short Term Histogram (LSTH):

- **Short-Term** (e.g., 1 hour): Captures recent request patterns, **Long-Term** (e.g., 1 day): Captures broader usage patterns.
- Combines short-term and long-term data for dynamic adjustments.

$$pre-warm = \gamma L_{prewarm} + (1-\gamma) S_{prewarm}$$

$$keep-alive = \gamma L_{keepalive} + (1-\gamma) S_{keepalive}$$

$\gamma = 0.5$

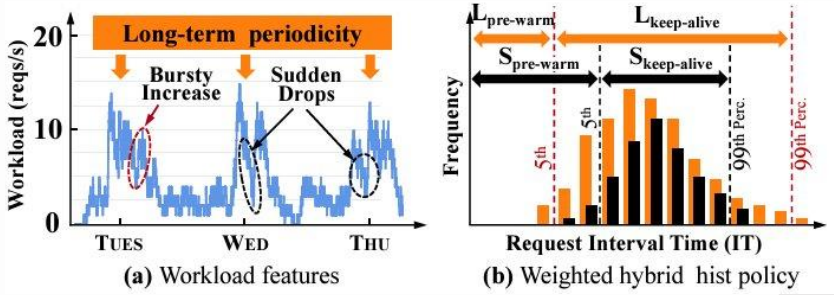


Fig: (a) The long-term periodicity and short-term burst behaviors of inference workloads. (b) The weighted hybrid hist policy by characterizing both long-term and short-term workload patterns.

Experimental Setup

Emulation Framework :

- **Purpose**: The experiment combines scale-up simulations with experiments on a local testbed cluster.

Testbed :

- Local cluster: 8-server cluster
- Large scale simulation: 2000-server cluster (No real execution in the cluster)

Workloads & Applications :

- Online second-hand vehicle trading (OSVT) : SSD, MobileNet, and ResNet-50, for object detection, license recognition and vehicle classification. SLO = 200ms
- Q&A robot service : TextCNN-69 , LSTM-2365 and DSSM-2389 for understanding user questions and finding matched answers. SLO = 50ms
- 3 production traces from azure functions : sporadic, periodic, brusty

Comparison with:

- **OpenFaaS$^+$:** original OpenFaaS + GPU support + other config.
- **BATCH** : A serverless inference system that adopts the OTP design

Component	Specification	Component	Specification
CPU device	Intel Xeon Silver-4215	Shared LLC Size	11MB
Number of sockets	2	Memory Capacity	128GB
Processor BaseFreq.	2.50 GHz	Operating System	Ubuntu 16.04
CPU Threads	32 (16 physical cores)	SSD Capacity	960GB
GPU device	Nvidia RTX 2080Ti	GPU Memory Config	11GB DGDDR6
GPU SM cores	4352	Number of GPUs	16

Table: Experimental testbed configuration

Evaluation: Throughput

- INFless increases the system, throughput by 5.2x and 2.6x times on average compared to baselines
- INFless benefits greatly from its resource scheduling algorithm because of the much fewer fragments
- Every component added on INFless contribute much for increasing throughput with batching contributing the highest

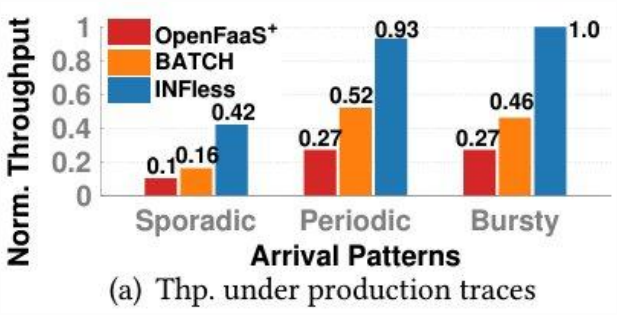


FIg: The normalized throughput comparison of INFless with baselines under different production traces

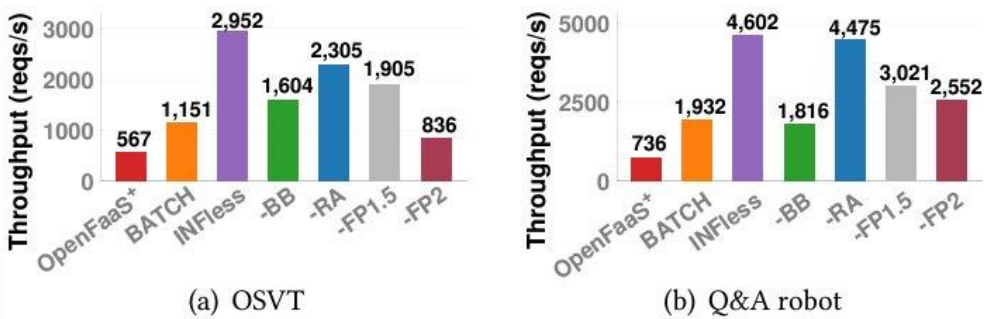


FIg:Throughput comparisons and component analysis of INFless, (constant request load)

Evaluation

Less SLO Violation

- INFless can guarantee the latency SLO of inference workloads.
- The SLO violation rate is less than 3.1% on average

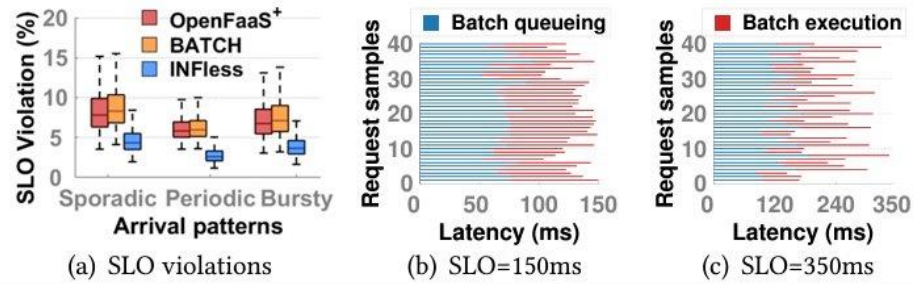


Fig: SLO violation comparison of INFless with base- lines and (b): latency breakdown of INFless under different latency SLO settings

Less over-provisioning

- INFless can reduce the provisioned resources by 60% compared to BATCH
- Request load declines, INFless can scale-in the number of instances quickly according to its flexible LSTH policy

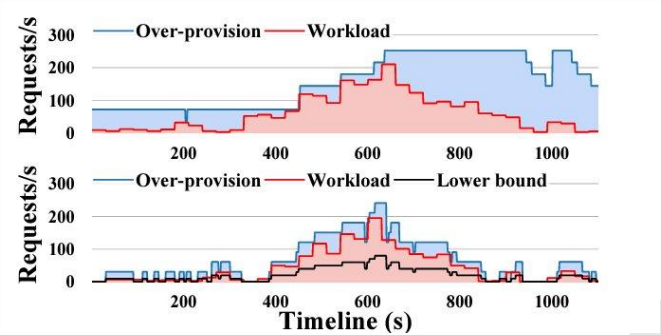
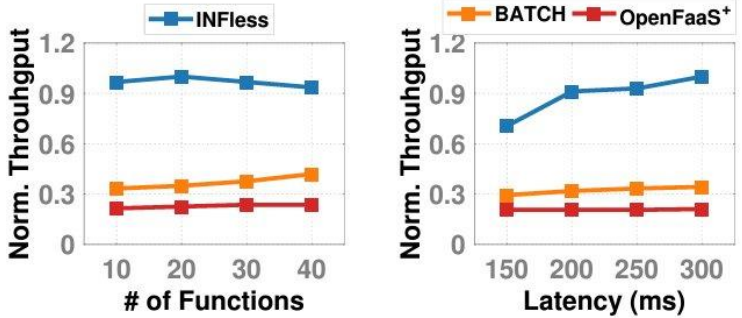


Fig: Resource provisioning by BATCH (top) and INFless (bottom)

Evaluation : Scalability & Cost Efficiency



(a) Thp. under different # of funcs (b) Thp. under different SLOs

Fig: : Scalability & Cost Efficiency

- When the concurrent request requests scales to 10000, INFless still achieves 2.6x and 4,2x higher throughput than BATCH and OpenFaas +

	AWS EC2	OpenFaas+	BATCH	INFless
CPUs per 100RPS	49.42	55.63	41.45	13.91
GPUs per 100RPS	2.47	2.13	1.34	0.51
Cost per request [$]	2.23×10^{-5}	2×10^{-5}	1.32×10^{-5}	1.6×10^{-6}

Fig: Computation cost comparison

- Considering the 400-server production cluster in 58.com with 1.9 billion requests per day, INFless could save about $1200000 everyday

Conclusion

- INFless introduces a native serverless platform specifically designed for machine learning inference, addressing the limitations of existing serverless solutions.
- It integrates built-in batching, GPU support, and non-uniform scaling to achieve low latency and high throughput.
- The Long-Short Term Histogram (LSTH) policy minimizes cold starts, improving resource efficiency.
- INFless outperforms traditional systems by 2-5x in throughput while meeting strict latency requirements.
- This makes INFless a practical solution for deploying inference models in real-world, latency-critical applications.
- Explore broader use cases beyond ML inference and further optimize resource efficiency.

Limitations

- **High Resource Usage:** INFless incurs higher resource costs, especially under bursty workloads, due to limited optimization in GPU-sharing and dynamic scaling.

- **SLO Challenges:** Struggles to maintain high SLO compliance in complex, multi-stage DNN workflows, where inter-function dependencies are critical.

Thanks!

Do you have any questions?