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

Yanan Yang
College of Intelligence & Computing (CIC), Tianjin University, Tianjin Key Lab. of Advanced Networking
Tianjin, China
ynyang@tju.edu.cn

Laiping Zhao*
CIC, Tianjin University, TANKLAB
Tianjin, China
laiping@tju.edu.cn

Yiming Li
CIC, Tianjin University, TANKLAB
Tianjin, China
l_ym@tju.edu.cn

Huanyu Zhang
CIC, Tianjin University, TANKLAB
Tianjin, China
3016218159@tju.edu.cn

Jie Li
CIC, Tianjin University, TANKLAB
Tianjin, China
lijie20@tju.edu.cn

Mingyang Zhao
CIC, Tianjin University, TANKLAB
Tianjin, China
mingyang@tju.edu.cn

Xingzhen Chen
58.com
Beijing, China
chenxingzhen@58.com

Keqiu Li
CIC, Tianjin University, TANKLAB
Tianjin, China
keqiu@tju.edu.cn

ABSTRACT

Modern websites increasingly rely on machine learning (ML) to improve their business efficiency. Developing and maintaining ML services incurs high costs for developers. Although serverless systems are a promising solution to reduce costs, we find that the current general purpose serverless systems cannot meet the low latency, high throughput demands of ML services.

While simply "patching" general serverless systems does not resolve the problem completely, we propose that such a system should natively combine the features of inference with a serverless paradigm. We present INFless, the first ML domain-specific serverless platform. It provides a unified, heterogeneous resource abstraction between CPU and accelerators, and achieves high throughput using built-in batching and non-uniform scaling mechanisms. It also supports low latency through coordinated management of batch queuing time, execution time and coldstart rate. We evaluate INFless using both a local cluster testbed and a large-scale simulation. Experimental results show that INFless outperforms state-of-the-art systems by 2×-5× on system throughput, meeting the latency goals of ML services.

CCS CONCEPTS

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

KEYWORDS

Serverless Computing, Machine Learning, Inference System

ACM Reference Format:
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. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507709

*Corresponding author: laiping@tju.edu.cn.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507709

1 INTRODUCTION

Due to its resource management-free, auto-scaling and cost efficiency advantages, serverless computing has been a success in multiple areas, e.g., web services, IoT monitoring applications, entertainment [16]. Machine learning (ML) inference, i.e., deploying well-trained models in applications to provide prediction or classification services, is yet another promising area for serverless adoption. Typical explorations include Amazon Alexa [4], Facebook Messenger Bot [28] and optical character recognition (OCR) [20].

Unlike scheduled background applications, ML inferences are often integrated into online websites (e.g., e-commerce, search engine, social network) and comes with strict latency requirements. For example, China's largest local life service website, relies on hundreds of ML inference services in their core business, ranging from advertisements, and question-answering robots, to fraud detection. Among these cases, 90% of the inference response times should be less than 200ms. Although prior works [3, 6, 41] have proved the great potential of serverless inference, current commercially available serverless services such as Amazon Lambda[25], Google Cloud Functions[19], and Azure Functions [5] do not cater to the needs of ML inference, as they do not address the challenge of providing solutions for guaranteeing latency, while the resource efficiency at

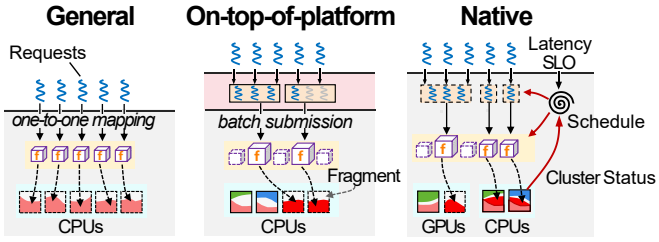


Figure 1: Schematic overview of serverless inference systems.

the serverless provider side is also very low; namely, (i) serverless providers do not allow users to specify latency requirements in their Service Level Objectives (SLOs) and therefore cannot satisfy diverse latency requirements, which may range from tens to hundreds of milliseconds across models [32], and (ii) large inference models are often highly parallelizable and computationally intensive, heavily relying on accelerators (e.g., GPUs, FPGAs or NPUs) for fast processing. However, they merely support the use of CPU resources.

To improve system throughput, recent frameworks such as Mark [41] and BATCH [3] adopt a batching method, which aggregates multiple inputs together for more efficient execution. However, their approaches all share an *On-Top-of-Platform* (OTP) design, i.e., *building another new buffer layer on top of the commercial serverless platform.* Although they do improve the cost-efficiency and throughput, the OTP design also introduces additional latency for request scheduling, and the end-to-end latency guarantee is still not addressed. Moreover, such designs are oblivious to the underlying resource allocations and have limited ability to optimize system throughput. For example, they are unable to adjust the resource allocation when the computational efficiency of a model varies over resource configurations and batchsizes; nor can they switch automatically between CPU and accelerators.

In this paper, we argue for a native serverless inference system, i.e., a serverless system tailored for inference serving by fully integrating the inference features inside the platform design (accelerator support, built-in batching, etc.). As shown in Figure 1, when compared with the OTP approach, the native design brings opportunities for exploring the full stack optimization from function scheduling to resource management, which enables it to take all operational responsibilities and provides high performance and resource efficiency for users and developers, respectively.

A native serverless inference system introduces several challenges that need to be addressed. **(i) Low latency:** In a serverless platform, the function scheduling delay, instance startup and batch processing will impact the end-to-end latency for serving a request. Managing the time overhead in each stage to guarantee overall latency is challenging. **(ii) High throughput:** Introducing accelerator support and a built-in batching mechanism complicates the resource management and function scheduling. How to choose the optimal hardware and batch configurations for runtime functions to maximize system throughput is also a challenge. **(iii) Low overhead:** The overall system should be practicable and efficient, and the newly added modules or modifications should also be lightweight and easy to use.

To overcome the abovementioned challenges, we build *INFless*, the first ML domain-specific serverless platform. *INFless* caters to inference as Backend-as-a-Service (BaaS) offerings. It accepts the function code of inference models and automates the deployment and scaling under varying workloads. *INFless* allows user to specify their high-level performance requirements (e.g., latency SLO). It can guarantee sub-second latency for user requests and achieve high resource efficiency through a *native design*, which fully combines the features of inference service with serverless paradigm.

Our contributions can be summarized as follows:

- We co-design the batch management and heterogeneous resource allocation mechanism, and propose the *non-uniform scaling* policy to maximize resource efficiency.
- We propose a lightweight *combined operator profiling* method that quickly infers an appropriate instance configuration to meet latency requirements.
- We design a novel *Long-Short Term Histogram* (LSTH) policy to reduce the cold start rate by 21.9%, while at the same time reduces resource waste by 24.3%.
- We completely implement *INFless* based on OpenFaaS [17], an open-source FaaS platform, and demonstrate its significant improvement in resource efficiency and SLO guarantee from both a real-world deployment and a large-scale of simulator evaluation.

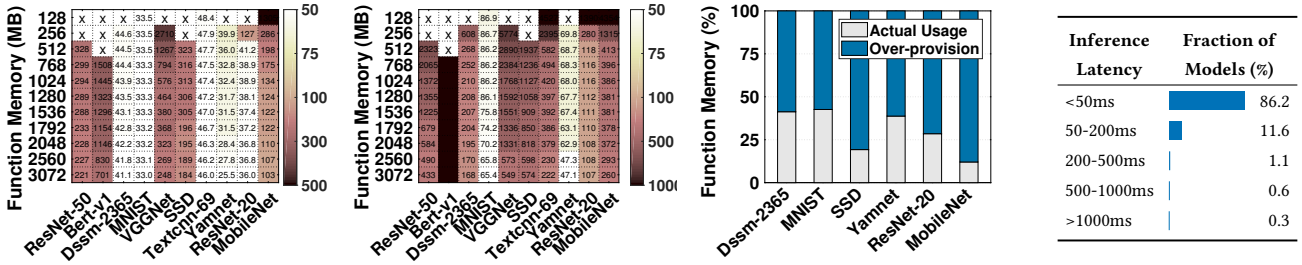
2 BACKGROUND & MOTIVATION

In this section, we show the inefficiency of inference on existing serverless platforms through an example study of AWS Lambda.

2.1 Serverless Inference

The serverless computing paradigm has undergone rapid growth over the past few years. Many applications are being deployed in commercial serverless platforms [4, 16, 20, 28, 36, 40]. In particular, the dominating application domain is web services [16], i.e., building serverless backends to handle web requests. Machine learning is also widely integrated into web services. For example, China's largest local life service website has deployed more than 600 ML inference models serving millions of requests every minute. These models include fraud detection, pornographic image recognition, false information detection and customer service robots. They are commonly structured as individual background modules. When users publish messages regarding topics such as housing rental, recruitment, secondhand products, catering, and entertainment on the website, these models are triggered at the backend to process them.

These inference services are commonly latency critical but have complex computations. For example, more than 90% of the models at the local life service website are required to respond within 200ms (Figure 2(d)). As cluster management and model configurations generate non-negligible cost for developers (especially under varied or bursty workloads [3]), serving inference on serverless systems becomes a promising solution. First, inference services could easily decouple from front-end applications and deploy as stateless functions. Second, developers could quickly build inference services using function templates without participating in instance management. Third, the auto-scaling ability of serverless computing could deal with bursty workloads well. Fourth, the pay-per-use billing model also saves money for service providers.



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

(a) Latency under no-batching (b) Latency under batching (c) Function memory over-provisioning (d) Latency SLO distribution

Figure 2: (a) The inference latency distribution when running models on AWS Lambda without batching support. (b) The inference latency distribution when running models on AWS Lambda with OTP-batching support, where × means that the memory size is too small to load the model. (c) The memory over-provisioning for achieving low latency requirement. (4) Real-world latency SLO distribution by the local life service website.

2.2 Limitations of Existing Serverless Platforms

Despite the great potential of serverless inference, after we study the performance of 11 typical inference models (Table 1) on AWS Lambda, we find that the current commercial serverless platform does not fit well with inference.

Observation #1: *High latency: The commercial serverless platform lacks the support of accelerators and therefore cannot provide low-latency services for large-sized inference models.*

Commercial serverless platforms often require developers to specify the memory size explicitly for their function instances (e.g., ranging from 128 MB to 3072 MB) and allocate CPU quotas in proportion to the memory size. We characterize the invocation time of inference models under different function memory sizes on AWS Lambda. To avoid the cold start latency, we continuously send inference requests to keep the function instance alive and measure the execution time for each invocation. As shown in Figure 2(a), the dark color represents high execution time, while the light color indicates better function performance, and × means the model cannot be loaded with too small a function memory size. We can see that small models (e.g., MNIST, Textcnn-69) can respond within 50ms under each memory configuration as long as it can be loaded, but for the other large models, such as Bert-v1, ResNet-50 and VGGNet, a small memory configuration leads to quite a long execution time (exceeding hundreds of milliseconds) due to the limited CPU quotas allocated to them. Even configured with the maximum allowable memory size, their execution time for a single request exceeds 200ms, making it difficult to meet the latency SLO in the production environment.

Observation #2: *For batch-enabled inference, commercial serverless platforms cannot provide low-latency services for some small-sized models.*

Batching is an optimization method specifically designed for ML inference. It can increase resource efficiency of inference services by processing multiple requests simultaneously [3, 13, 37]. To evaluate the inference performance under batching, we implement the OTP batching mechanism[3] on AWS Lambda. We set the batchsize to 4 or 8 and measure the invocation time by varying the allocated memory size. Figure 2(b) shows that nearly all inference models experience poor performance under batching. In particular, for models including DSSM-2365, SSD, Textcnn-69 and MobileNet, batching increases their execution time by more than 4×, so that the latency exceeds 200ms, making it difficult for them to provide low-latency inference services for users.

Table 1: ML inference models collected from the MLPerf benchmark and real-world commercial services.

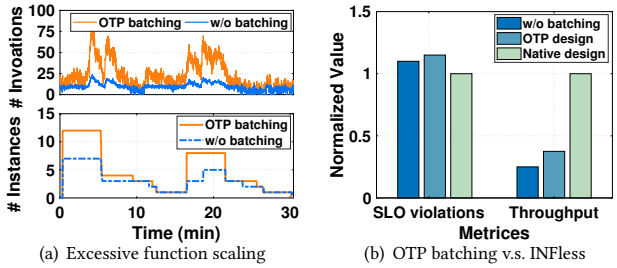
ML Model	Network Size	GFLOPs	Description & Source
Bert-v1	391M	22.2	Language processing[12]
ResNet-50	98M	3.89	Image classification[23]
VGGNet	69M	5.55	Feature localisation[33]
LSTM-2365	39M	0.10	Text Q&A system[9]
ResNet-20	36M	1.55	Image classification[23]
SSD	29M	2.02	Object detection[8]
DSSM-2365	25M	0.13	Text Q&A system[2]
YamNet	17M	1.60	Speech recognition[27]
MobileNet	17M	0.05	Mobile network[42]
TextCNN-69	11M	0.53	Text classification[7]
MNIST	72k	0.01	Number recognition[18]

Observation #3: *Resource over-provisioning: The proportional CPU-memory allocation policy set by a commercial serverless platform does not fit with computationally-intensive inference. It encourages over-provisioning of memory, resulting in poor resource utilization.*

Commercial serverless providers such as AWS Lambda and Google Cloud Functions allocate CPU power in proportion to the amount of memory configured. Developers can increase or decrease the memory and CPU power allocated to the function using the *memory* setting. Although AWS Lambda can provide a latency of < 200ms for some of the models (e.g., SSD and MobileNet under no-batching in Figure 2(a) or ResNet-20 and DSSM-2365 under batching in Figure 2(b)), we find that its *proportional CPU-memory allocation policy* tends to apply for a larger memory to obtain a sufficient CPU quota. As shown in Figure 2(c), more than 50% of the function memory is over-provisioned for serving these models to meet the latency SLO. For example, serving SSD without batching within 200ms requires at least 1,792 MB of function memory allocation, while the actual consumption is only 427 MB. This leads to much cloud resource waste and increases user cost under the current serverless pricing models[40].

Observation #4: *The "one-to-one mapping" request processing policy of commercial serverless platforms causes low resource utilization.*

Existing commercial platforms generally only support a "one-to-one mapping" policy for processing requests; i.e., each inference request is dispatched to a separate instance. This policy inherently causes an excessive number of instances to be created, especially under bursty workloads. While it may not be problematic for other workloads, the excessive instances are supposed to be reduced through batching for ML workloads. Figure 3(a) shows the number



(a) Excessive function scaling (b) OTP batching v.s. INFless

Figure 3: (a) Excessive instances created by the "one-to-one" mapping policy. (b) Comparison of performance and throughput between the OTP batching method and INFless.

of instances activated for serving ResNet-20 with and without OTP batching. By aggregating the user requests into batches (we set *batchsize* = 4), the total number of function invocations declines by 72%. As an instance may be reused by a later-arriving request, the total number of launched instances under batching also declines by 35%. Moreover, the memory usage decreases from 117,555 GB·s to 96,303 GB·s.

Observation #5: *OTP batching lacks the codesign of batch configuration, instance scheduling and resource allocation, bringing only limited throughput improvement.*

The prior OTP batching strategy [3] supports aggregating requests into batches before submitting them to AWS Lambda. This design improves the throughput and reduces the monetary cost. However, there are some drawbacks to it: (1) OTP batching has to be deployed and maintained at another dedicated server, generating great cost for developers. (2) It is unaware of the scheduling delay and queuing time inside the serverless platform, which makes it hard to meet the low latency requirements of many inference functions. (3) It can only select instance configurations outside of AWS Lambda, and the *uniform scaling* policy in these platforms make it unable to adaptively tune parameters when the workload varies, leading to suboptimal scheduling and resource provisioning decisions. Figure 3(b) shows that although OTP batching improves throughput by 30% compared with the original AWS Lambda, our native serverless inference system *INFless*, which determines the batch configuration, instance scheduling and resource allocation collaboratively, could further improve throughput by nearly 3× compared with the OTP batching method.

2.3 Implications

The limitations of existing serverless platforms make it particularly desirable to have a novel, native serverless inference system that is able to provide both low latency guarantees and high throughput. To achieve this goal, the serverless inference system should be able to address the observations above: (1) *Supporting hybrid CPU/accelerators*: Due to the computationally intensive nature of inference models (**Observations 1 and 2**), it is necessary to introduce the support of accelerators in serverless platforms to provide low latency for inference models (especially for large-sized models). (2) *Producing resource-efficient scheduling*: Instead of the proportional CPU-memory allocation policy (**Observation 3**), the serverless inference system should support flexible resource allocation for improving the throughput. Optimal scheduling can be approximated closely by knowing the resource demand profile of the inference models. (3) *Supporting built-in batching*: While batching is able to improve system throughput (**Observation 4**), OTP batching with the uniform scaling policy in existing serverless platforms is still inefficient (**Observation 5**); the serverless inference system should support built-in batching and explore more flexible instance scaling design for higher throughput.

However, it is challenging to design such a system. The control ability of the backend system has to be enhanced significantly, and it commonly requires full-stack optimizations across both software and hardware layers. We begin our exploration from addressing the following challenges: (1) The hardware affinity and interchangeability [26] complicate the management of hybrid CPU/accelerators; (2) The batchsize and resource selection further enlarge the search space of scheduling decisions; (3) The running overhead should also be low to make the system more practicable.

3 METHODOLOGY & SYSTEM DESIGN

In this section, we present the design of *INFLess*.

3.1 System Architecture

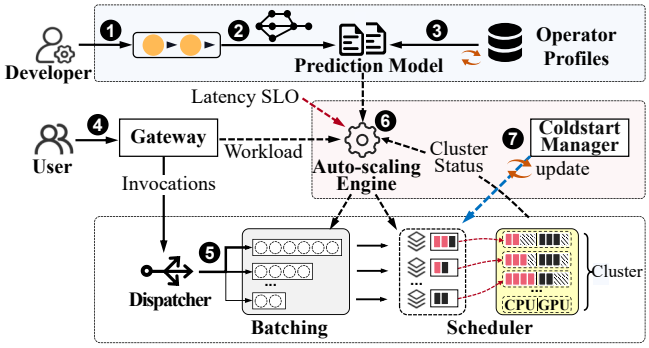


Figure 4: The design overview of *INFless*.

The basic idea is that *a serverless inference platform should exploit the features and characteristics of inference (e.g., shared operators, batching and computation intensive) to optimize system performance.* Figure 4 highlights the design overview of *INFless*. A core component of *INFless* is the *non-uniform scaling* engine. Instances of the same function may have different configurations, which enables flexible allocation of resources and workloads at instance level. The autoscaling engine takes the inference model profiles and cluster resource status as input, explores to determine instance configurations (batchsize, CPU-GPU quotas, request arrival rate and placement) to maximize the resource efficiency under an SLO guarantee.

INFless is positioned as a domain-specific serverless platform for easy and cost-efficient inference service development. For developers, *INFless* provides function templates to help them easily build inference functions (Figure 5). When a function is deployed❶, *INFless* builds a performance prediction model by combining the profiles of its operators. This method is lightweight since it only requires parsing the DAG structure of the model❷, and the overall latency of a model can be easily estimated through a combination of its operators' execution times. As inference models often share

the same set of operators [1], we only need to profile the operators beforehand❸.

For a user's inference queries❹, *INFless* automatically generates instance configurations that meet workload and latency requirements. The arrived requests pack into built-in batches and dispatch to the corresponding instances❺. If the workload or cluster resource status changes, the *auto-scaling* engine adaptively tunes the new instance configurations with a non-uniform scaling policy by selecting from the optimized batch-resource decisions to best match the system goals❻. As instance cold start generates non-negligible latency, *INFless* further designs an LSTH policy to avoid the start latency ❼. Overall, the latency of a request consists of three parts, $l = t_{cold} + t_{batch} + t_{exec}$, where t_{cold} is the cold start time and $t_{cold} = 0$ if cold start is avoided, t_{batch} is the time of waiting in the batch, and t_{exec} represents the batch execution time.

```
provider:
  name: INFless
  gateway: http://127.0.0.1:31112
functions:
  resnet-50:
    lang: python3
    handler: ./resnet-50
    image: resnet50-faas:latest
    latency_target: 100ms #SLO
    requests:
      memory: 4096Mi #optional
      cpu: 2         #optional
...
```

```
...
...
    labels:
      com.infless.max.batch: 8
    environment:
      MODEL_NAME: resnet-50
      MODEL_BASE_PATH: /models
      BATCH_BASE_PATH: /batches
    configuration:
      copy:
        - ./script
        - ./models
        - ./batches
```

Figure 5: Template fields of INFless

3.2 Built-in, Non-Uniform Batching

Unlike the uniform batching in OTP method [3], *INFless* adopts a batching mechanism that is both *built-in* and *non-uniform*:

Built-in: Batching is integrated into the serverless platform, enabling simultaneous, collaborative control over batchsize, resource allocation and placements.

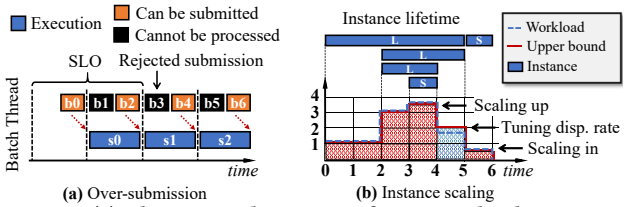
Non-uniform: Each instance has an individual batch queue to aggregate inference requests. The batchsize and resource quotas of each instance, regardless of whether it is from the same function, can differ and enable full utilization of resource fragments in the cluster.

A user request first waits in a batch queue until the batch is entirely filled or times out. Clearly, a high request arrival rate could fill the batch queue quickly, and requests have to be dropped if the previous batch is already full but not yet submitted (Figure 6(a)). To guarantee the latency SLO (i.e., $l \le t_{slo}$) without dropping any requests, the request arrival rate toward each instance is strictly kept within a range of $[r_{low}, r_{up}]$, where r_{low} (or r_{up}) denotes the lower (or upper) bound of the workload that can be processed in one instance. Supposing the cold start can be avoided, and denoting by b the batchsize of the instance, we have

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

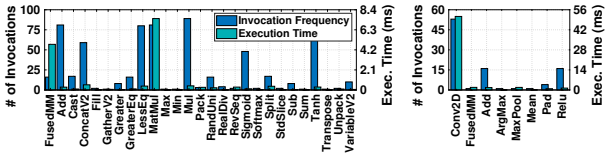
Note that we have $t_{exec} \le t_{slo}/2$ for ensuring $r_{low} \le r_{up}$, i.e., the batch submission speed should not exceed the batch execution speed. For example, given a latency SLO of 200ms, if the execution time is 50ms for a function instance whose batchsize is 4, then the workload dispatched to this instance should be in the range of $[28, 80]$ requests per second (RPS).

Suppose there already exist n instances for an inference function in the system. Let $R_{max} = \sum_{i \in [1,..,n]} r^i_{up}$ and $R_{min} = \sum_{i \in [1,..,n]} r^i_{low}$.



(a) Over-submission **(b)** Instance scaling

Figure 6: (a) The over-submission of requests lead to request drop. (b) An example of the instance scaling procedure, and L (S) represents function instance configured with large (small) batchsize, respectively.



(a) LSTM-maxclass (b) ResNet-50

Figure 7: Calling frequency and execution time of the DNN operators. (a) LSTM-maxclass contains 27 operators, and *MatMul* takes more than 95% of the overall execution time; (b) ResNet-50 contains 8 operators, and most of the execution time is spent on *Conv2D*.

Denoting by R the actual overall RPS towards the function, then the RPS for each instance is controlled as follows:

(i) $R > R_{max}$: In this case, each instance will be dispatched with an RPS as high as r^i_{up}. Then, the *auto-scaling* engine will be notified to launch new instances for processing the residual RPS (i.e., $R - R_{max}$).

(ii) $\alpha R_{min} + (1-\alpha)R_{max} \le R \le R_{max}$: To avoid frequent scaling oscillation under workload fluctuations, we allow each instance to have a varied RPS which is controlled using a constant $\alpha \in [0, 1]$. That is, the RPS dispatched to an instance i is in proportion to its range size: $r_i = r^i_{up} - \frac{R_{max} - R}{R_{max} - R_{min}} \times (r^i_{up} - r^i_{low})$. In practice, we prefer a request arrival rate of an instance to be as close to its upper bound as possible to improve throughput (Figure 6(b)), and α is set to 0.8 in our implementation.

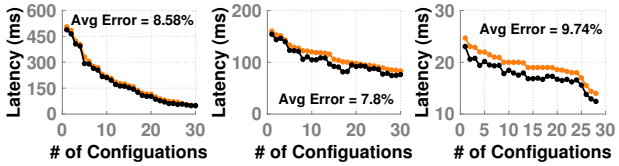
(iii) $R < \alpha R_{min} + (1-\alpha)R_{max}$: In this case, the *auto-scaling* engine will release extra instances to return to the above case, and the workload dispatched to each active instance is also re-calculated.

3.3 Combined Operator Profiling

INFless improves system throughput by deploying as many instances as possible with limited resources. However, insufficient resource allocation to an instance would lead to high latency, resulting in SLO violation. To guarantee the SLO, we design a prediction model to estimate the latency under various batchsizes and resource configurations. As offline profiling every inference function is rather costly, especially in our application scenario where hundreds of inference models are deployed or updated every day, our prediction model instead adopts a lightweight *combined operator profiling* (COP) method.

Observation #6: *Inference functions share a common set of operators, and the execution time is dominated by a small subset of them.*

Inference functions can be structured as a number of connected operators [1]. We summarize the 11 models in Table 1 and find that although there are more than 1,000 calls of operators in these



(a) ResNet-50 (b) MobileNet (c) LSTM-2365

Figure 8: The prediction results of operator combination model across different batch-resource configurations.

models, the number of distinct operators is only 71. Figure 7 shows the number of occurrences of the 35 operators in Lstm-2365 and ResNet-50. In Lstm-2365, the *MatMul* operator (i.e., matrix multiplication) is called 81 times, while the *Sum* operator (i.e., vector summation) appears only once. In addition, we observe that the overall execution time is dominated by only a small subset of operators. As shown in Figure 7(a), *FusedMatMul* and *Matmul* take up 76% of the overall execution time, and operators such as *ConcatV2* and *Mul* only take up less than 5%. Similarly, more than 95% of the ResNet-50 execution time is spent on *Conv2d* (Figure 7(b)). Hence, a model profile can easily be estimated by the dominated operators.

Given an operator o_i, we define its profile as a 5-tuple $o_i = \langle p_i, b_i, c_i, g_i, t_i \rangle$, where p_i represents the size of each piece of input data (e.g., the size of an image); b_i represents the batchsize; c_i represents the CPU-related resources, including cores, memory bandwidth, LLC, memory, etc; g_i represents the GPU-related resources, including GPU memory, SMs, PCI-e bandwidth, etc; and t_i is the corresponding execution time under the above configuration. Thus far, we have collected more than 100 operators' profiles and stored them in an operator profile database. Due to the massive number of combinations of p_i, b_i, c_i and g_i, we merely consider some discrete values in their separate feasible ranges. For example, the batchsize is selected from $b_i \in \{2^0, 2^1, ..., 2^{max}\}$, where 2^{max} denotes the maximum allowable batchsize for the model.

Operator profiles can be combined to estimate the overall inference latency under various batchsizes and resource configurations. Let $G = (O, E)$ be the task graph constructed by the set of operators O and their dependency relations E. The graph can be deconstructed into two basic structures, including a *sequence chain* and *parallel branches*. For a *sequence chain*, the execution time of the chain (t_{chain}) equals to the sum of all its operators. That is,

$$t_{chain} = \sum_{i \in \{chain\}} t_i$$

For *parallel branches*, the overall execution time of the branches ($t_{branches}$) equals to the maximum time among all branches. That is,

$$t_{branches} = \max_{i \in \{branches\}} t_i$$

In the case of a task graph with a combined structure of *sequence chains* and *parallel branches*, the overall execution time can be estimated through a combination of t_{chain} and $t_{branches}$.

Figure 8 shows the prediction error $|\hat{P} - P|/P$ of inference latency across different batch and resource configurations, where $\hat{P}$ and P denote the predicted batch execution time and the actual value, respectively. The operator combination model achieves less than 10% of the average prediction error for different inference functions. For example, the prediction errors of ResNet-50, MobileNet and LSTM-2365 are 8.6%, 7.8% and 9.74%, respectively. LSTM-2365 has the highest average prediction error of 9.74% since it has more execution paths overlapping in its operator DAG workflow. In practice, we choose to increase the prediction offset by 10% to reduce the risk of SLO violations from prediction errors.

3.4 Scheduling

The *auto-scaling* engine monitors the real-time RPS and judges if the existing instances are sufficient to fulfill these requests. If not, it dispatches part of requests to existing instances and calls the scheduling algorithm to launch new instances for processing the residual workload. Given the inference performance prediction model and the request arrival rate, the *scheduling* module explores the optimal configurations of function instances to minimize resource usage while guaranteeing their latency SLOs.

Denoted by m the number of available servers in the cluster. Suppose there are at most n instances (either from the same or different inference functions) to schedule. For an instance i, $i \in [1, .., n]$, we need to determine its configurations, including b_i, c_i, g_i, and its schedule x_{ij} (i.e., a binary variable that has the value "1" if instance i is scheduled on server j; otherwise, it is set to "0"). If an instance is configured with $b_i = 0$, then it has never launched. We further define a binary variable y_j, which has the value "1" if server j is used for instance deployment. Otherwise, it is set to "0". Hence, the optimization problem can be formulated as follows:

$$minimize: \quad \sum_{j}^{m} (\beta C_j + G_j) y_j \tag{2}$$

$$t^i_{wait} + t^i_{exec} \le t^i_{slo}, \quad \forall i \in [1, .., n] \tag{3}$$

$$t^i_{exec} \le t^i_{wait}, \quad \forall i \in [1, .., n] \tag{4}$$

$$\sum_{i}^{n} c_i x_{ij} \le C_j y_j, \quad \forall j \in [1, .., m] \tag{5}$$

$$\sum_{i}^{n} g_i x_{ij} \le G_j y_j, \quad \forall j \in [1, .., m] \tag{6}$$

$$\alpha R^k_{max} + (1 - \alpha) R^k_{min} \le R_k \le R^k_{max}, \quad \forall k \in I \tag{7}$$

$$x_{ij} \in \{0, 1\}, \quad y_j \in \{0, 1\} \tag{8}$$

$$b_i, c_i \in Z_+, \quad g_i \in Z \tag{9}$$

Objective (2) defines the hybrid CPU and GPU resources occupied by the instances. In particular, C_j and G_j denote the available CPU and GPU resources in server j. Hence, this objective includes the fragments on each server that cannot be used. Since CPU and GPU resources are not directly comparable, we solve this problem using a conversion factor β and evaluate the best β by comparing the floating point operations per second (FLOPS) of the two types of resources. Constraints (3) and (4) ensure that the latency SLO is satisfied. In particular, the batch execution time $t^i_{exec} = f(b_i, c_i, g_i)$ can be derived through the *combined operator profiling* prediction model and the waiting time $t^i_{wait} = b_i / r_i$, where r_i is the request arrival rate toward instance i. Constraints (5) and (6) ensure that the CPU and GPU resources allocated to the instances do not exceed the available resources on each server. Constraint (7) and Equation (1) together ensure that the arrived requests R_k toward function k can be fully processed by all of its launched instances. Constraints (8) and (9) refer to the domain constraints.

Algorithm 1: Schedule(R_k, $\boldsymbol{B}$, $\boldsymbol{M}$, t_{slo})

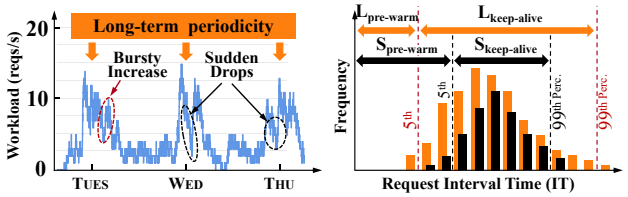
Input:
R_k ▷ The residual RPS towards the function k;
$\boldsymbol{B}$ ▷ The batchsize set of k, sorted in the descending order;
$\boldsymbol{M}$ ▷ The available resource capacities for the m-server cluster
t_{slo} ▷ The latency SLO for function k;
Output:
n_k ▷ The number of instances for function k;
b_i, c_i, g_i ▷ The batchsize/CPU/GPU configs. of instances;
x_{ij} ▷ the placement of each instance;

1 $n_k \leftarrow 0, x_{ij} \leftarrow 0$;
2 **while** $R_k > 0$ **do**
3 **for** $b \in \mathbf{B}$ **do**
4 $\boldsymbol{I}_b \leftarrow$ `AvailableConfig`(b, R_k, t_{slo})
5 /* e.g., $\mathbf{I}_b = \{\langle b, c_1, g_1\rangle, \cdots, \langle b, c_n, g_n\rangle\}$ */
6 **if** $I_b = NULL$ **then**
7 **continue;** // try next largest batchsize
8 **for** $\forall \langle b, c_i, g_i\rangle \in \mathbf{I}_b, \forall j \in \mathbf{M}$ **do**
9 Derive the res. efficiency e_{ij} using Equation 10;
10 **if** $\{e_{ij}\} \neq NULL$ **then**
11 $i, j \leftarrow argmax\{e_{ij}\}, \forall i \in [1..n], j \in [1..m]$;
12 $n_k \leftarrow n_k + 1, x_{ij} \leftarrow 1$; // schedule i to j
13 $\langle C_j, G_j\rangle \leftarrow \langle C_j, G_j\rangle - \langle c_i, g_i\rangle$;
14 $R_k \leftarrow R_k - r_{up}$;// update R_k
15 **break;** // scaling for the rest of R_k
16 **Function** `AvailableConfig`(b, R_k, t_{slo}):
17 $\boldsymbol{I}_b \leftarrow NULL$;
18 **for** $\forall \langle b, c_i, g_i\rangle \in all\_configurations$ **do**
19 $t_{exec} \leftarrow f(b, c_i, g_i)$;// predict the t_{exec}
20 **if** $b = 1$ **then**
21 **if** $t_{exec} \leq t_{slo}$ **then**
22 $\boldsymbol{I}_b \leftarrow \boldsymbol{I}_b \cup \{\langle b, t_i, g_i\rangle\}$;
23 **else**
24 Derive the r_{up} and r_{low} using Equation 1;
25 **if** $t_{exec} \leq t_{slo}/2 \wedge R_k \geq r_{low}$ **then**
26 $\boldsymbol{I}_b \leftarrow \boldsymbol{I}_b \cup \{\langle b, c_i, g_i\rangle\}$;
27 **return** $\boldsymbol{I}_b$

This optimization problem is at least as hard as the known NP-hard bin packing problem [37]. Therefore, we resort to a greedy scheduling algorithm.

Algorithm 1 shows the details of our scheduling algorithm. As the batchsize is one of the key components that contributes most to throughput (§ 5.2), we always first explore the configurations with a larger batchsize for each new instance. The algorithm iteratively checks whether the maximum batchsize can be employed in lines 2-15. Given a candidate batchsize b, the algorithm invokes the function *AvailableConfig()* to explore all possible resource configurations (denoted by $\boldsymbol{I}_b$) that meet the latency SLO. We check whether $t_{exec} \leq t_{slo}/2$ and $R_k \geq r_{low}$ are established simultaneously (the former ensures that Equations (3) and (4) are satisfied, and the later ensures the batches are saturated before the waiting timeout). If yes, the current configuration of $\langle b, c_i, g_i\rangle$ is feasible, and we add it into the set $\boldsymbol{I}_b$ (Lines 24-26). Note that there is no batch queuing time when the batchsize is 1, and we need only to check whether t_{exec} meets the latency SLO in lines 20-22. If there are no feasible resource configurations with the batchsize b, the algorithm returns to line 3 and continuously checks the next largest batchsize (Lines 6-7). Otherwise, the algorithm selects the new instance's resource configuration from $\boldsymbol{I}_b$ and launches it on a server (Lines 8-15).

In the selection process, as we would like to maximize the throughput while reducing the resource fragmentation as much as possible, we derive a resource efficiency metric (e_{ij}) for every combination



(a) Workload features **(b)** Weighted hybrid hist policy

Figure 9: (a) The long-term periodicity and short-term burst behaviors of inference workloads. (b) We propose the weighted hybrid hist policy by charactering both long-term and short-term workload patterns.

of instance configuration $\langle b, c_i, g_i\rangle$ and server j as,

$$e_{ij} = \frac{RPS/resource}{fragmentation} = \frac{r_{up}/(\beta c_i + g_i)}{1 - (\beta c_i + g_i)/(\beta C_j + G_j)} \quad (10)$$

where the numerator ($RPS/resource$) is computed as a normalized score between $[0, 1.0]$. We select the combination with the maximum e_{ij} and schedule the instance on server j. We then update the available resources on server j and the residual RPS (Lines 13-14) and return to line 1 for scaling the next instance (Line 15). Note that we omit the memory constraint in the scheduling algorithm design since the memory usage of these inference models are much smaller than the memory capacity of our servers. However, it can be easily extended to cover more resource dimensions using resource vectors.

3.5 Managing Cold Starts with LSTH

Cold starts cause significant performance degradation for serverless functions [39]. Especially for inference functions, it is even higher than the actual query execution time due to the large-sized models and serving library [22]. Thus, it is critical to reduce the number of cold starts to avoid SLO violations. A state-of-the-art approach is the *hybrid histogram policy* (HHP) [36], which tracks the idle times of a configurable duration (e.g., 4 hours) and draws a histogram to derive two parameters: (1) *pre-warming window*: The time the policy waits, since the last execution, before it loads the function image expecting the next invocation; (2) *keep-alive window:* The time during which a function's image is kept alive after it has been loaded to memory [36]. However, after we apply HHP into our inference web service scenarios, we find that it is so conservative that it generates too much resource waste.

Figure 9(a) shows a 3-day request load towards a fraud detection model at the local life service website. We see that the request load exhibits two distinctive features: (1) *Long-term periodicity (LTP): the request load shows a diurnal user access pattern overall;* (2) *Short-term burst (STB): there are many sudden changes (including both increases and decreases) in short times.* While LTP enables the request predictability, STB reduces the prediction accuracy. In HHP, as the parameters of *pre-warming* and *keep-alive* heavily rely on the data collected in the tracked duration, setting an appropriate duration length is rather critical. However, in case of a request load with both LTP and STB features (Figure 9(a)), this becomes a significant challenge. *A long duration leads to a conservative setting of* pre-warming *(i.e., a smaller pre-warming window), resulting in resource waste when the RPS suddenly decreases. Conversely, a short duration*

cannot capture the periodicity of requests, making the histogram not representative and increasing the cold start rate.

To address this problem, we propose a *Long-Short Term Histogram* (LSTH) policy to derive the *pre-warming* and *keep-alive*, which can reduce the resource waste while avoiding cold starts. As shown in Figure 9(b), we track both the long-term and short-term application idle times and draw two histograms. The two histograms represent the request patterns in the last short (e.g., 1 hour) and long durations (e.g., 1 day), respectively. Like HHT, we identify the head (e.g., the $5th$ percentile) and tail (e.g., the $99th$ percentile) of the idle time distribution and use the head to select the pre-warming window for the application, and the tail to select the keep-alive window. The head and tail generated from the short-term histogram are denoted by $S_{prewarm}$ and $S_{keepalive}$, and the head and tail generated from the long-term histogram are denoted by $L_{prewarm}$ and $L_{keepalive}$. We derive the *pre-warming* and *keep-alive windows* using their weighted sum. That is, $pre-warm = \gamma L_{prewarm} + (1-\gamma)S_{prewarm}$ and $keep-alive = \gamma L_{keepalive} + (1-\gamma)S_{keepalive}$, where γ is a configurable weight between 0 and 1; by default, we set $\gamma = 0.5$.

In case of a sudden spike that exceeds the peak request arrival rate in history, LSTH will not be able to prepare sufficient number of instances and coldstart inevitably occurs. We can use techniques like SOCK [30] and Catalyzer[15] to accelerate the functions' startup.

4 SYSTEM IMPLEMENTATION

INFless is implemented based on OpenFaaS [17], an event-driven serverless computing platform atop Kubernetes, with approximately 9,300 lines of Golang. More than 94% of the code was used to modify the original modules or add new functional components in *faas-netes*, and the rest was used for enhancing *faas-cli* and *faas-Gateway*. For the other modules including authority certification, security check and NATs streaming, we opted to reuse them in *INFless*. The prototype system is available here [1]. Additionally, we have developed approximately 5,000 lines of code (Java, Python and Linux Shells) for the system testing, simulation and load generation.

INFless's auto-scaling engine is fully integrated into the *scalingHandler* of *faas-netes* to replace its original scheduling module. We make a full stack of functional modifications within OpenFaaS so that *INFless* can cooperate with them. For example, we introduce GPU support into the container runtime layer. By using the container access interface to CUDA devices provided by NVIDIA-docker [14], *INFless* can access and manage GPU resources after mounting its drivers and libraries into the container. To achieve high resource utilization and performance isolation, we adopt the *CUDA MPS* technique[29] for partitioning the GPU streaming multiprocessors and use *linux cgroups* for binding CPUs to different instances. We further use TensorFlow serving [31] for deploying the user's ML models and create a batch queue inside the container when the instance is initialized.

At the application layer, we modify the *ParseYAML()* in *faas-cli* to provide the SLO declaration interface to users. At the system layer, our work mainly includes (i) developing a *register repository* to store data including the function profiles, instance configurations and cluster available resources; (ii) adding a new *predictor* module in *faas-netes* for predicting the execution time using the combined operator profiling model; (iii) modifying the original trigger rules in *scalingAlert* and implementing the batch-aware dispatching mechanism inside the *request dispatcher*; and (iv) replacing the scheduling algorithm and *coldstart manager* with Algorithm 1 and LSTH.

To help developers build inference services quickly, *INFless* provides function templates in Python 2.7/3.4. It currently supports the direct development for inference models compiled by TensorFlow. We have also developed a format conversion tool to make it compatible with models created by Keras[10] and Pytorch[34].

[1]https://github.com/TankLabTJU/INFless/

5 EVALUATION

5.1 Methodology

Table 2: Experimental testbed configuration.

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

Experimental setup: Our experiment combines scale-up simulations with experiments on a local testbed cluster.

In **local cluster experiments**, we use a 8-machine cluster equipped with 16 Nvidia RTX 2080Ti GPUs. Table 2 summarizes the configurations of the cluster. The machines are connected via 10 Gbps, full-bisection bandwidth Ethernet.

In **simulations**, we programmatically scale out the cluster to 2,000 servers to evaluate the effectiveness of controller algorithms and overhead in *INFless*. The simulator runs INFless's real code and scheduling logic against the simulated machines. There are three important differences to note. First, the simulated function invocations are used only for workload arrival rate collection and are not be forwarded to any instance for execution. Second, the auto-scaling engine just makes scheduling decisions and records the cluster and function status without creating or deleting any instance. Third, we collect only the theoretical throughput upper bound and scheduling overhead as the evaluation metrics.

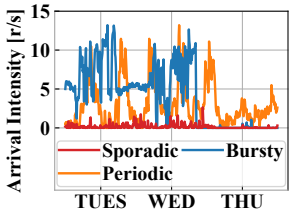
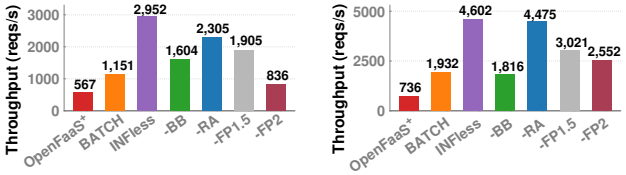


Figure 10: The three production trace examples.

Workloads: We create two practical applications using the models in Table 1, according to the *online secondhand vehicle trading* (OSVT) business and *Q&A robot* service at the local life service website. The OSVT employs machine learning models including SSD [8], MobileNet [42] and ResNet-50 [23] for object detection, license recognition and vehicle classification. Its latency SLO is set to 200ms. The *Q&A robot* uses TextCNN-69 [7], LSTM-2365 [9] and DSSM-2389[2] for understanding user questions and finding matched answers. Its latency SLO is set to 50ms. Their batchsizes are set to $\leq$ 32. These inference services are triggered by both constant and dynamic invocations. The dynamic invocations are simulated using the production trace from Azure Function [36], which include 7-day request statistics with both LTP and STB features. Figure 10 shows parts of the three typical types of production traces in [36]: *sporadic*, *periodic* and *bursty*.

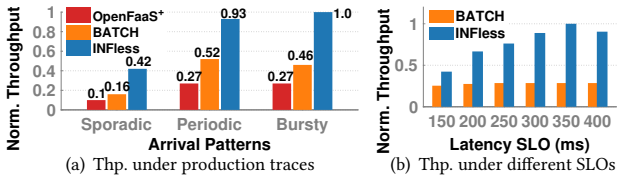
Table 3: Comparison of serverless inference systems.

Features	OpenFaaS+	BATCH	INFless
GPU devices support	Yes	Yes	Yes
Batching mechanism	No	OTP	Built-in
Function profiling	No	Yes	Combined Ops
Instance auto-scaling	Uniform	Uniform	Non-Uniform
Batch-aware dispatcher	No	No	Yes
Keep-alive policy	Fixed	Fixed	Dynamic



(a) OSVT (b) Q&A robot

Figure 11: Throughput comparisons and component analysis of INFless. BB: build-in, non-uniform batching; RS: resource scheduling; OP1.5: add the predicted latency by 50%; OP2: add the predicted latency by 100%.



(a) Thp. under production traces (b) Thp. under different SLOs

Figure 12: The normalized throughput comparison of INFless with baselines: (a) under different production traces; (b) under different latency SLOs.

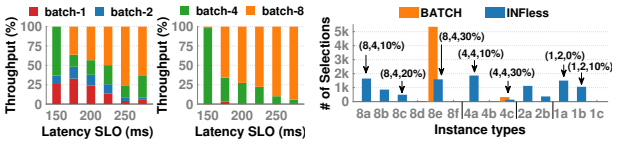
Comparison systems: We evaluate *INFless* with serverless frameworks *OpenFaaS*+ and *BATCH*. Table 3 shows the comparison among them.

OpenFaaS+: We enhance the original *OpenFaaS* to support GPU for a fair comparison. As it does not support batching, we simply configure each of its instances with 2 CPU cores and 10% GPU SMs and set its fixed keep-alive window as 300 seconds.

BATCH [3]: A serverless inference system that adopts the OTP design. It proposes an adaptive batching method to reduce the inference cost. Since the original *BATCH* is implemented atop AWS Lambda, we redevelop it atop OpenFaaS and extend its memory-only function profiles with CPU and GPU allocations.

5.2 Local Cluster Evaluation

High throughput: *INFless* improves system throughput by 2×-5×. Given the limited cluster resources, we run stress testing on *OpenFaaS*+, *BATCH* and *INFless* using a constant request load. Figure 11 shows the maximum RPS achieved by them in both OSVT and Q&A robot scenarios. We can see that *INFless* can improve the throughput by 5.2× and 2.6× on average compared with that of *OpenFaaS*+ and *BATCH*, respectively. In Figure 12(a), we further evaluate the throughput (i.e., the RPS divided by its occupied resources) using the three types of production workload. We see that *INFless* can improve the throughput by 4.3×, 3.4× and 3.6× on average compared with *OpenFaaS*+, and by 2.6×, 1.8× and 2.2× on average than *BATCH*, under *sporadic*, *periodic* and *bursty* loads, respectively. Although the *sporadic* workload results in many cold



(a) INFless (b) BATCH (c) Instance configuration selection

Figure 13: Throughput distribution contributed by different batchsize settings by (a) INFless and (b) BATCH. (c) Resource configuration distribution of instances by INFless and BATCH. (b, c, g) represents the batchsize, CPU core and GPU SM configuration of instances.

starts, *INFless* still achieves a throughput 3× higher than that of *BATCH*. Relaxing the SLO can also help improve the throughput. Figure 12(b) shows that *INFless* achieves 1.6×-3.5× higher throughput than *BATCH* under various SLO settings for OSVT. *INFless* benefits greatly from its resource scheduling algorithm because of the much fewer fragments, whereas *BATCH*'s throughput does not increase much due to its resource waste.

Component analysis: Every component of *INFless* contributes much to throughput improvement, with batching being the highest. *INFless* employs multiple novel techniques for improving the throughput, including *built-in non-uniform batching* (BB), *combined operator predicting* (OP), and *resource scheduling* (RS). We evaluate their separate contributions by measuring the throughput decrease after ablating each one. In particular, BB is disabled by setting all batchsizes =1; RS is disabled by selecting only the resource configuration with the maximum throughput; OP is disabled by simply adding the predicted latency by 50% (OP1.5) and 100% (OP2).

As shown in Figure 11, when we disable the BB, OP and RS in the OSVT scenario, the throughput drops by 45.6%, 35.4% and 21.9%, respectively. In the Q&A robot scenario, it drops by 60%, 34.3%, 7%, respectively. BB contributes the most to the throughput improvement. Reducing the prediction accuracy from OP1.5 to OP2 makes *INFless* conservatively choose smaller-batch instances in scaling decisions, and this also makes it more easily under-estimate the instance's upper bound capacity (r_{up}), resulting in much higher resource waste. Disabling RS decreases the throughput by only 7% in the case of Q&A robot, which is different from the 21.9% drop in the OSVT scenario. This is because the inference models in Q&A robot service are small-in-size, and their instances generate few resource fragments even without RS.

Flexible configurations: *INFless* opts for flexible configurations on both batchsizes and resource allocations. Although a large batchsize can improve resource efficiency, it is effective only when the RPS is high, otherwise the batch cannot saturate before the SLO violation. The non-uniform batching policy enables *INFless* to select small batchsizes under the low RPS, so that resource fragmentation is reduced. Figures 13(a) and 13(b) show the throughput distribution contributed by different batchsize settings for ResNet-50. While *BATCH* mainly utilizes 2 types of batchsizes (4 and 8), the batchsize settings of *INFless* are flexibly chosen from {1, 2, 4, 8}. Figure 13(c) depicts the resource configuration distribution of instances for ResNet-50. We see that *INFless* also opts for various resource allocation settings, no matter what the batchsize is, whereas *BATCH* uses only three configurations. This demonstrates that *INFless* can fully exploit the potential of resources.

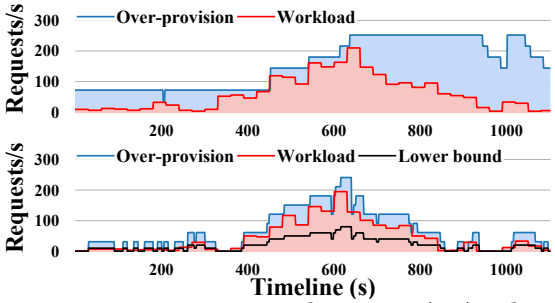
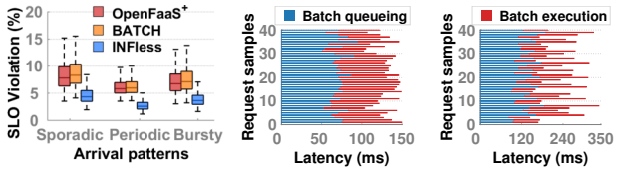


Figure 14: Resource provisioning by BATCH (top) and INFless (bottom).



(a) SLO violations (b) SLO=150ms (c) SLO=350ms

Figure 15: (a): SLO violation comparison of INFless with baselines and (b): latency breakdown of INFless under different latency SLO settings.

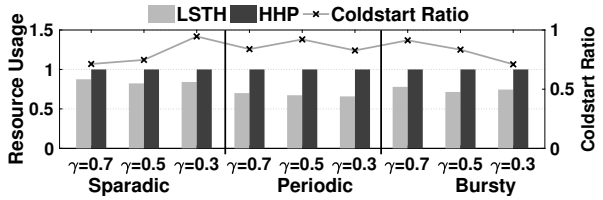
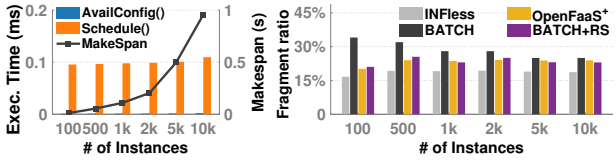


Figure 16: Cold start rate comparison.



(a) Scheduling overhead (b) Resource fragments

Figure 17: Scheduling overhead and resource fragments of INFless in large-scale simulations.

Less over-provisioning:*INFless*'s resource allocation policy reduces the resource provisioning significantly. Taking ResNet-50 as an example, Figure 14 shows the resource provisioning over a period by *BATCH* and *INFless*, respectively. Since *BATCH* always prefers a larger batch (Figure 13(b)), it provisions more resources than *INFless* during the increase of the request load. Whenever the request load declines, *INFless* can scale-in the number of instances quickly according to its flexible LSTH policy, whereas *BATCH* still holds its resources for some time due to its fixed keep-alive policy. In this example, *INFless* can reduce the provisioned resources by 60% compared with *BATCH*.

SLO violation: *INFless* can guarantee the latency SLO of inference workloads. Figure 15(a) shows that the SLO violation rate by *INFless* is $\leq$ 3.1% on average, which is far lower than that of *OpenFaaS*$^+$ and *BATCH*. In fact, since *OpenFaaS*$^+$ adopts the "one-to-one" request mapping policy for launching instances, its execution time is much lower than those of *BATCH* and *INFless*. However, as its fixed-keeplive policy for cold start management generates a much higher cold start rate, the overall SLO violation rate increases–e.g., reaching 8% under the *sporadic* load. *BATCH* has a similar SLO violation rate to OpenFaaS$^+$, due to its batch queuing timeout setting. *INFless* is able to guarantee more than 95% of request latency SLO even in the sporadic (more cold start invocations) and bursty (higher scaling pressures) workload patterns. Figures 15(b) and 15(c) show the latency breakdown by *INFless* when setting the latency SLO = 150*ms* and = 350*ms*, respectively. *INFless* can regulate the queueing time roughly equal to the execution time.

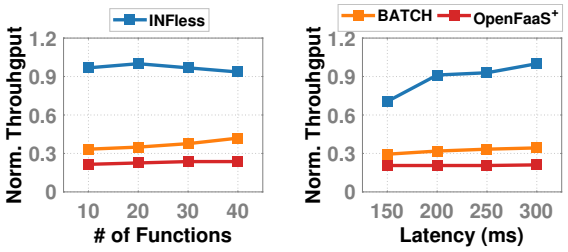
Cold start: Compared with *HHP*, our *LSTH* policy can reduce the cold start rate by 20%. We evaluate the cold start rate by *LSTH* under the various production traces and the parameter settings (i.e., $\gamma \in \{0.3, 0.5, 0.7\}$) in Figure 16. We set the LTP duration of LSTH to 24 hours and the STB duration to 1 hour. We see that the cold start rate by *LSTH* is 21.9% lower than that of *HHP*, and its idle resource waste is reduced by 24.3% on average. Moreover, we can obtain the lowest resource waste when $\gamma = 0.5$.

5.3 Large Scale Simulation

Scalability: *INFless* scales well in large-scale evaluations. To evaluate *INFless* under different number of functions and instances, we create no more than 40 functions by varying their respective SLOs and request loads and launch up to 10,000 instances in the simulations. Figure 17(a) shows the scheduling overhead generated by the *Schedule()* algorithm. *Schedule()* is highly efficient, so scheduling a single instance takes only 0.5ms. When the concurrent requests scales to 10,000, the overall scheduling overhead is still less than 1 second. Figure 18(a) shows the throughput per unit of resource generated by the three systems. We see that *INFless* still achieves 2.6× and 4.2× higher throughput than *BATCH* and *OpenFaaS*$^+$. By fixing the number of functions at 20 and increasing the latency from 150ms to 300ms, we find that the throughput per unit of resource by *INFless* also increases from 0.7 to 1.0 due to the less lower resource allocation to each instance (Figure 18(b)).

Resource fragments: *INFless*'s resource-aware scheduling algorithm reduces the resource fragments significantly. We measure the amount of unallocated resources in each active server and derive the resource fragment ratio by dividing it by all the server's resources. Figure 17(b) shows the average resource fragment rate generated by the three systems. To evaluate the effectiveness of the *scheduling algorithm*, we further pass the instances configured by *BATCH* to the algorithm, obtaining a fourth system *BATCH+RS*. We see that *INFless* generates a resource fragment ratio as low as 15%, which is much lower than the others. *BATCH+RS* also performs a lower fragment ratio than *BATCH*, demonstrating the effectiveness of the *scheduling algorithm*.

Cost efficiency: *INFless* can help service developers and cloud providers reduce the cost of constructing inference services. To evaluate the economic benefits of *INFless*, we record the CPU and GPU consumptions and derive the average computation cost per inference request. We set the price of a CPU to 0.034$/hour, following the setting of the r5.2×large service at AWS EC2. Since



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

Figure 18: Throughput evaluation in the large-scale simulation.

Table 4: Computation cost comparison.

	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}

AWS EC2 does not provide instances with an NVIDIA GeForce 2080Ti, we refer to the pricing model of Tesla P100 in p3.2×large and transform it into the price of 2080Ti GPUs equaling 2.5$/hour. Table 4 shows that *INFless* reduces the cost per request by > 10× compared to that by AWS EC2 and OpenFaaS$^+$. Considering the local life service website that deploys more than 600 inference models serving 1.9 billion requests per day (more than 20,000 RPS), they currently rely on a cluster with 400 servers, and the billing cost approximates $4,253 per day. If we move all these inference services into *INFless*, the billing cost will drop to $941 per day, saving approximately $1,200,000 every year.

6 RELATED WORK

Inference on Serverless: Serving ML models on serverless systems has been explored in several recent works [3, 6, 21, 38, 41]. MArk [41] explored the cost effectiveness of serverless inference and proposed a hybrid approach of using both AWS EC2 and a serverless system for inferences, where the serverless system is responsible for handling arrival bursts. Swayam [21] is an auto-scaling framework deployed atop Microsoft Azure, which aims to minimize the resource waste for ML inference by using a predictive provision model. BARISTA [6] presents a scalable serving system using the serverless system to reduce resource provision based on the workload prediction. Cloudflow [38] focuses on the prediction serving pipeline on the serverless system and provides optimizations including function fusion and competitive execution to improve the performance. BATCH [3] (the state-of-the-art) designs a buffer layer on top of the serverless platform and bundles requests together with batching for cost-saving serverless inference. Although they have promoted the development of serverless inference, they still cannot meet the low-latency, high-throughput demands of modern websites.

Adaptive batch/resource tuning: Tuning batch or resource configuration adaptively for improving performance has been studied in [11, 13, 35]. Clipper [11] introduces caching, batching, and adaptive model selection techniques to reduce the latency. GSLICE [13] adopts a heuristic to select the suitable batch and resource configurations. INFaaS [35] could automatically choose the model variant, batchsize and hardware according to users' accuracy and performance requirements. These works provide insights for performance optimization but have not considered adaption in serverless computing.

Hybrid CPU/GPU management: Existing serverless platforms [5, 17, 19, 24, 25] lack unified hardware abstractions and support only the allocation of CPU/memory resources. Inference platforms such as Nexus [37], GSLICE [13] and INFaaS [35] support the GPU provision but neglect the coordination between them. AlloX [26] features the interchangeability of CPU-GPU resources and places ML applications on the right resources to maximize efficiency. However, it does not consider the challenges in serverless computing.

7 CONCLUSION

E-commerce, social networks, search engines, etc., increasingly rely on ML services. Deploying inference services on serverless systems can reduce the overall cost for developers due to its auto-scaling engine and pay-per-use billing model. We examine the current general purpose serverless platforms and find that they cannot satisfy the low-latency, high-throughput requirements of inference services. Hence, we design *INFless*, which adopts batching and hybrid CPU-GPU resources and can improve the throughput significantly while guaranteeing their latency SLO. We evaluate the effectiveness of *INFless* using two real application scenarios from a local life service website. Experimental results demonstrate the massive cost savings for developers.

In the future, we would like to further study and optimize the performance of inference function chains in the serverless platform.

ACKNOWLEDGMENTS

We thank our shepherd Mike Marty and anonymous ASPLOS reviewers for their extremely insightful comments and suggestions that have significantly improved the quality of this paper. We also thank Dr. Tao Li for his significant contribution to this work. This work is supported by the National Natural Science Foundation of China under grant 61872265, 62141218; the new Generation of Artificial Intelligence Science and Technology Major Project of Tianjin under grant 19ZXZNGX00010; State Key Lab. of Computer Architecture (ICT, CAS)-CARCHA202009; NUDTPDL-6142110200405. It is also supported by Meituan.

A ARTIFACT APPENDIX

A.1 Abstract

We implement the INFless system in OpenFaas v0.18.13, and run it on a Kubernetes v1.18.3 cluster. INFless is designed to be deployed by cloud providers and provides FaaS-based inference services for different user from AI scenarios, just like the existing serverless platforms such as AWS Lambda. The source code of INFless has been released in GitHub (see the guidance of https://github.com/TankLabTJU/INFless/). With INFless, the inference models uploaded by the developer could be easily deployed as inference functions. INFless automatically manages these functions including the resource allocation, scaling and instance scheduling. Once an inference function is deployed successfully, it could be invoked as individual service or embedded into web applications as backend modules.

To help to reproduce the experimental results, we have uploaded the materials for system setup and function deployment. The inference models are stored in developer directory and the load generator generates requests towards benchmarks from AI scenarios in 58.com. The scheduling decisions and instance activities are logged by INFless, and the experimental results are plotted with Matlab. More details are listed as below. ‘

A.2 Artifact check-list (meta-information)

- **Program:** Linux kernel 4.18.0, Docker 19.03-ce, Kubernetes v1.18.3.
- **Compilation:** go1.13.15.
- **Data set:** function trace from Micro Azure.
- **Run-time environment:** Ubuntu 16.04.
- **Hardware:** Intel x86 servers connected via 10 Gbps, full-bisection bandwidth Ethernet.
- **Experiments:** Inference applications from OSVT (Online Secondhand Vehicle Trading) and Q&A robot scenarios.
- **How much disk space required (approximately)?:** 50GB per server to store instance image file.
- **How much time is needed to prepare workflow (approximately)?:** 2 hours.
- **How much time is needed to complete experiments (approximately)?:** 2+ hours.
- **Publicly available?:** Yes.
- **Workflow framework used?:** No.

A.3 Description

A.3.1 How to access. The source code, scripts, and instructions are available on GitHub: https://github.com/TankLabTJU/INFless/.

A.3.2 Hardware dependencies. The INFless implementation works on Intel x86 CPU servers. The CPU-GPU scheduling experiments additionally require GPUs to have with support for CUDA MPS technique support (e.g., Nvidia RTX 2080Ti GPUs).

A.3.3 Software dependencies. The components of INFless is deployed on Kubernetes cluster with NVIDIA-docker which supports the CUDA device usage inside containers, and Linus *cgroups* is also needed to bind CPUs to different instances for isolation.

A.3.4 Data sets. We use the production trace from Azure Function, which include 7-day request statistics with three different workload arrival patterns: sporadic, periodic and bursty.

A.3.5 Models. We use two practical applications according to the online secondhand vehicle trading (OSVT) business and Q&A robot service at the local life service website. The OSVT employs machine learning models including SSD, MobileNet and ResNet-50 for object detection, license recognition and vehicle classification. The Q&A robot uses TextCNN-69, LSTM-2365 and DSSM-2389 for understanding user questions and finding matched answers.

A.4 Installation

Users need to install INFless on a Kubernetes cluster with Intel x86 machines, the servers are used for running inference functions. Additionally, user need to prepare a client machine to generate workloads for inference serving. Ubuntu 14.04 or 16.04 is recommended for each server.

To build and install INFless framework, user need to download the source code and scripts from GitHub. The components of INFless should firstly be compiled and deployed in the kubernetes master node. The inference model profiling metadata and cluster configuration files should also be placed in the master node (directory of '/root/yaml/') before launching INFless system. In the following, we list the directory structure of the GitHub files:

- **README.md** This file has detailed instructions to run INFless.
- **sourceCode** The component source code of INFless for compiling and installation.
- **configuration** The cluster configuration files for launching INFless.
- **developer** The inference functions and deployment scripts with INFless.
- **models** The metadata of used inference models.
- **profiler** The model's profiling metadata.
- **workload** The function workload trace used for evaluation.
- **scripts** One brief instruction for the evaluation reproduction.

Source Code Compilation: After downloading the source code and scripts from GitHub, user need to compile the components of INFless and deploy them. The source code of *gateway* and *faas-netes* is in the directory of *sourceCode/Go*. The necessary commands for compiling INFless are listed as follows:

```
Switch into the directory of Go
# cd sourceCode/Go/
# cd src/github.com/openfaas/faas/gateway
# make
.....building.....
Successfully built 6f15a4cf589d
Successfully tagged openfaas/gateway:latest-dev

# cd src/github.com/openfaas/faasnetes
# make
.....building.....
Successfully built 6f12c4cf323d
Successfully tagged openfaas/faas-netes:latest-dev
```

When the components are compiled, using the following commands to deploy INFless into kubernetes cluster.

```
Switch into the directory of Go
# cd sourceCode/Go/
# cd src/github.com/openfaas/faas-netes
# kubectl apply -f yaml/inuse
```

Building Inference Function Image: The function image of INFless is built based on TensorFlow Serving framework (i.e., the base image). We provide a pre-configured base image *sdcbench/tfseving-infless:latest* in DockerHub for building user's own inference functions. TensorFlow Serving and NVIDIA-docker is supported in base images, which enables the inference model to use both CPU and accelerators. To obtain the base image, please pull the base image from DockerHub to master node and tag it using the following command:

```
# docker pull sdcbench/tfseving-infless:latest
# docker tag sdcbench/tfseving-infless:latest
tensorflow/serving:latest-gpu
```

Function Deployment: Once INFless is launched, user can use the *faasdev-cli* tool on the server machine to upload and build inference functions, so they can be set up to run experiments. The *faasdev-cli* tool provides several options: *faasdev-cli build* is to build the user's function images using the base image. *faasdev-cli deploy* is to deploy the function into INFless, *faasdev-cli list* and *faasdev-cli delete* are used for listing functions and deleting them, respectively. Commands for compiling the *faasdev-cli* tool are listed as follows:

```
# cd INFless/sourceCode/Go/
# cd src/github.com/openfaas/faas-cli
# make
.....building.....
Successfully built 6f15a4cf589d
# cp faasdev-cli /usr/local/bin
Deploy inference functions # cd INFless/developer/serving-Functions
# faasdev-cli build -f resnet-50.yml
# faasdev-cli deploy -f resnet-50.yml
```

Notice: More details for deploying inference models could be found in GitHub (INFless/developer/README.md).

Workload Generator Installation: To conduct experiments, user need to deploy inference functions inside INFless platform and set up the *loadGen* from client node. Using the following commands to compile and install *loadGen*:

```
Switch into the directory of Java
# cd sourceCode/Java/
# cd loadGen
# mvn package -Dmaven.skip.test=true
BUILD SUCCESS
Total time: 22.650 s
Finished at: 2021-07-28T10:43:37+08:00
```

Notice: More details please see INFless/workload/README.md.

In addition, the *loadGen* also needs *LoadGenSimClient.jar* to generate function invocations. To generate workload for a deployed inference functions inside INFless, we should deploy the *loadGen* tool as a web service, and then use the *LoadGenSimClient.jar* tool to connect it with JAVA RMI for controlling the request arrival rate. The *loadGen* is compiling as a *loadGen.war* file and should be deployed with Apache Tomcat, the recommended Java version is JDK 1.8.0 or latest ones. Once the *loadGen* is successfully started, it will expose server port 22222 for the *LoadGenSimClient*, then we could use the following commands to generating workloads for the deployed functions.

```
Starting the workload generator
# cd INFless/workload/
# jps -l |grep Load |awk 'print $1' |xargs kill -9
# sh start_load.sh 192.168.1.109 22222
Collecting results
# cd workload
# sh start_load.sh
Baseline: INFless
Total statistics QPS:x
Scaling Efficiency: x
Throughput Efficiency: x
```

Notice: More details please see INFless/scripts/README.md.

Experimental Results: The experimental results are plotted with Matlab (INFless/sourceCode/Matlab). The details are listed as following:

(1) Figure 2(a): *INFless/motivation/heat/LantencyHeatmap_nobatch.m*
(2) Figure 2(b): *INFless/motivation/heat/LantencyHeatmap_batch.m*
(3) Figure 2(c): *INFless/motivation/hist/memoyConfigWasteBar.m*
(4) Figure 3(a): *INFless/motivation/BATCH_comparation_workload_instanceNum_subfig.m*
(5) Figure 3(b): *INFless/motivation/hist/comparison_batch_with_oneToOne.m*
(6) Figure 7(a): *INFless/method/lstm_OperatorsCount.m*
(7) Figure 7(b): *INFless/method/resnet_OperatorsCount.m*
(8) Figure 8(a): *INFless/evaluation/operatorDAGModel/Resnet50.m*
(9) Figure 8(b): *INFless/evaluation/operatorDAGModel/Mobilenet.m*
(10) Figure 8(c): *INFless/evaluation/operatorDAGModel/Lstm2365.m*
(11) Figure 11(a): *INFless/evaluation/throughput/trafficAblation.m*
(12) Figure 11(b): *INFless/evaluation/throughput/qsSystem.m*
(13) Figure 12(a): *INFless/evaluation/throughput/overall_normalized_throughput_across_workload_bar.m*
(14) Figure 12(b): *INFless/evaluation/throughput/overall_normalized_throughput_across_SLOs_all_bar.m*
(15) Figure 13(a): *INFless/evaluation/throughput/batch_configurations_across_SLOs_INFLess.m*
(16) Figure 13(b): *INFless/evaluation/throughput/batch_configurations_across_SLOs_BATCH.m*
(17) Figure 13(c): *INFless/evaluation/throughput/batch_configurations_details.m*
(18) Figure 15(a): *INFless/evaluation/box/SLO_violoation_box.m*
(19) Figure 15(b): *INFless/evaluation/hist/latency_partition_150ms_barh.m*
(20) Figure 15(c): *INFless/evaluation/hist/latency_partition_350ms_barh.m*
(21) Figure 16: *INFless/evaluation/hist/Coldstart_bar.m*
(22) Figure 17(a): *INFless/evaluation/hist/scheduling_latency_bar.m*
(23) Figure 17(b): *INFless/evaluation/hist/fragement_ratio_bar.m*
(24) Figure 18(a): *INFless/evaluation/line/Throughput_simulation.m*
(25) Figure 18(b): *INFless/evaluation/line/Throughput_simulation_slo.m*

A.5 Materials

Available source code: https://github.com/TankLabTJU/INFless/

REFERENCES

[1] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David M. Brooks. 2016. Fathom: reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25-27, 2016*. IEEE Computer Society, 148–157. https://doi.org/10.1109/IISWC.2016.7581275

[2] AILab. 2019. *Deep learning inference*. https://github.com/wuba/dl_inference.

[3] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 69, 15 pages.

[4] AWS. 2020. Serverless Application Lens: Alexa Skills. https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/alexa-skills.html.

[5] Microsoft Azure. 2020. Azure Functions—Serverless Architecture. https://azure.microsoft.com/en-us/services/functions/.

[6] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. 2019. BARISTA: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services. *CoRR* abs/1904.01576 (2019). arXiv:1904.01576 http://arxiv.org/abs/1904.01576

[7] YuKun Cao and Tian Zhao. 2018. Text Classification Based on TextCNN for Power Grid User Fault Repairing Information. In *5th International Conference on Systems and Informatics, ICSAI 2018, Nanjing, China, November 10-12, 2018*. IEEE,

1182–1187. https://doi.org/10.1109/ICSAI.2018.8599486

[8] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. 2018. DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *IEEE Trans. Pattern Anal. Mach. Intell.* 40, 4 (2018), 834–848. https://doi.org/10.1109/TPAMI.2017.2699184

[9] Liu Chen, Guangping Zeng, Qingchuan Zhang, Xingyu Chen, and Danfeng Wu. 2017. Question answering over knowledgebase with attention-based LSTM networks and knowledge embeddings. In *16th IEEE International Conference on Cognitive Informatics & Cognitive Computing, ICCI*CC 2017, Oxford, United Kingdom, July 26-28, 2017*, Newton Howard, Yingxu Wang, Amir Hussain, Freddie Hamdy, Bernard Widrow, and Lotfi A. Zadeh (Eds.). IEEE Computer Society, 243–246. https://doi.org/10.1109/ICCI-CC.2017.8109757

[10] François Chollet. 2015. Keras. https://keras.io.

[11] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[13] Aditya Dhakal, Sameer G. Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 492–506. https://doi.org/10.1145/3419111.3421284

[14] NVIDIA Docker. 2018. https://github.com/NVIDIA/nvidia-docker/.

[15] 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 *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 467–481. https://doi.org/10.1145/3373376.3378512

[16] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2020. A Review of Serverless Use Cases and their Characteristics. *CoRR* abs/2008.11110 (2020). arXiv:2008.11110 https://arxiv.org/abs/2008.11110

[17] A. Ellis. 2020. OpenFaas. https://docs.openfaas.com/.

[18] Google. 2018. Tensorflow serving benchmarks. https://github.com/tensorflow/serving.

[19] Google. 2020. Google Cloud Functions. https://cloud.google.com/functions/.

[20] GoogleCloud. 2020. Serverless Optical Character Recognition (OCR) Tutorial. https://cloud.google.com/functions/docs/tutorials/ocr.

[21] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. 2017. Swayam: distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017*, K. R. Jayaram, Anshul Gandhi, Bettina Kemme, and Peter R. Pietzuch (Eds.). ACM, 109–120. https://doi.org/10.1145/3135974.3135993

[22] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. 2020. Fifer: Tackling Resource Underutilization in the Serverless Era. In *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, Dilma Da Silva and Rüdiger Kapitza (Eds.). ACM, 280–295. https://doi.org/10.1145/3423211.3425683

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[24] IBM. 2020. Openwhisk. https://www.ibm.com/cloud-computing/bluemix/openwhisk.

[25] AWS Lambda. 2015. Serverless Compute - Amazon Web Services. https://aws.amazon.com/lambda/.

[26] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. AlloX: compute allocation in hybrid clusters. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 31:1–31:16. https://doi.org/10.1145/3342195.3387547

[27] Peter Mattson, Hanlin Tang, Gu-Yeon Wei, Carole-Jean Wu, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David A. Patterson, and Guenther Schmuelling. 2020. MLPerf: An Industry Standard Benchmark Suite for Machine Learning Performance. *IEEE Micro* 40, 2 (2020), 8–16. https://doi.org/10.1109/MM.2020.2974843

[28] Philipp Muens. 2020. Serverless Facebook Messenger Bot. https://github.com/pmuens/serverless-facebook-messenger-bot.

[29] NVIDIA. 2018. NVIDIA Turing Architecture Whitepaper. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[30] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[31] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. *CoRR* abs/1712.06139 (2017). arXiv:1712.06139 http://arxiv.org/abs/1712.06139

[32] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Shanker Khudia, James Law, Parth Malani, Andrey Malevich, Nadathur Satish, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim M. Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. *CoRR* abs/1811.09886 (2018). arXiv:1811.09886 http://arxiv.org/abs/1811.09886

[33] Omkar M. Parkhi, Andrea Vedaldi, and Andrew Zisserman. 2015. Deep Face Recognition. In *Proceedings of the British Machine Vision Conference 2015, BMVC 2015, Swansea, UK, September 7-10, 2015*, Xianghua Xie, Mark W. Jones, and Gary K. L. Tam (Eds.). BMVA Press, 41.1–41.12. https://doi.org/10.5244/C.29.41

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library

[35] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. INFaasCache: Keeping serverless computing alive with greedy-dual caching: Managed & Model-less Inference Serving. *CoRR* abs/1905.13348 (2019). arXiv:1905.13348 http://arxiv.org/abs/1905.13348

[36] Mohammad Shahrad, Rodrigo Fonseca, Iñigo 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 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[37] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 322–337. https://doi.org/10.1145/3341301.3359658

[38] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E. Gonzalez, and Joseph M. Hellerstein. 2020. Optimizing Prediction Serving on Low-Latency Serverless Dataflow. *CoRR* abs/2007.05832 (2020). arXiv:2007.05832 https://arxiv.org/abs/2007.05832

[39] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[40] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with ServerlessBench. (2020). https://doi.org/10.1145/3419111.3421280

[41] 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. https://www.usenix.org/conference/atc19/presentation/zhang-chengliang

[42] You Zhou, Yiyue Liu, Guijin Han, and Yiping Fu. 2019. Face Recognition Based on The Improved MobileNet. In *IEEE Symposium Series on Computational Intelligence, SSCI 2019, Xiamen, China, December 6-9, 2019*. IEEE, 2776–2781. https://doi.org/10.1109/SSCI44817.2019.9003100