

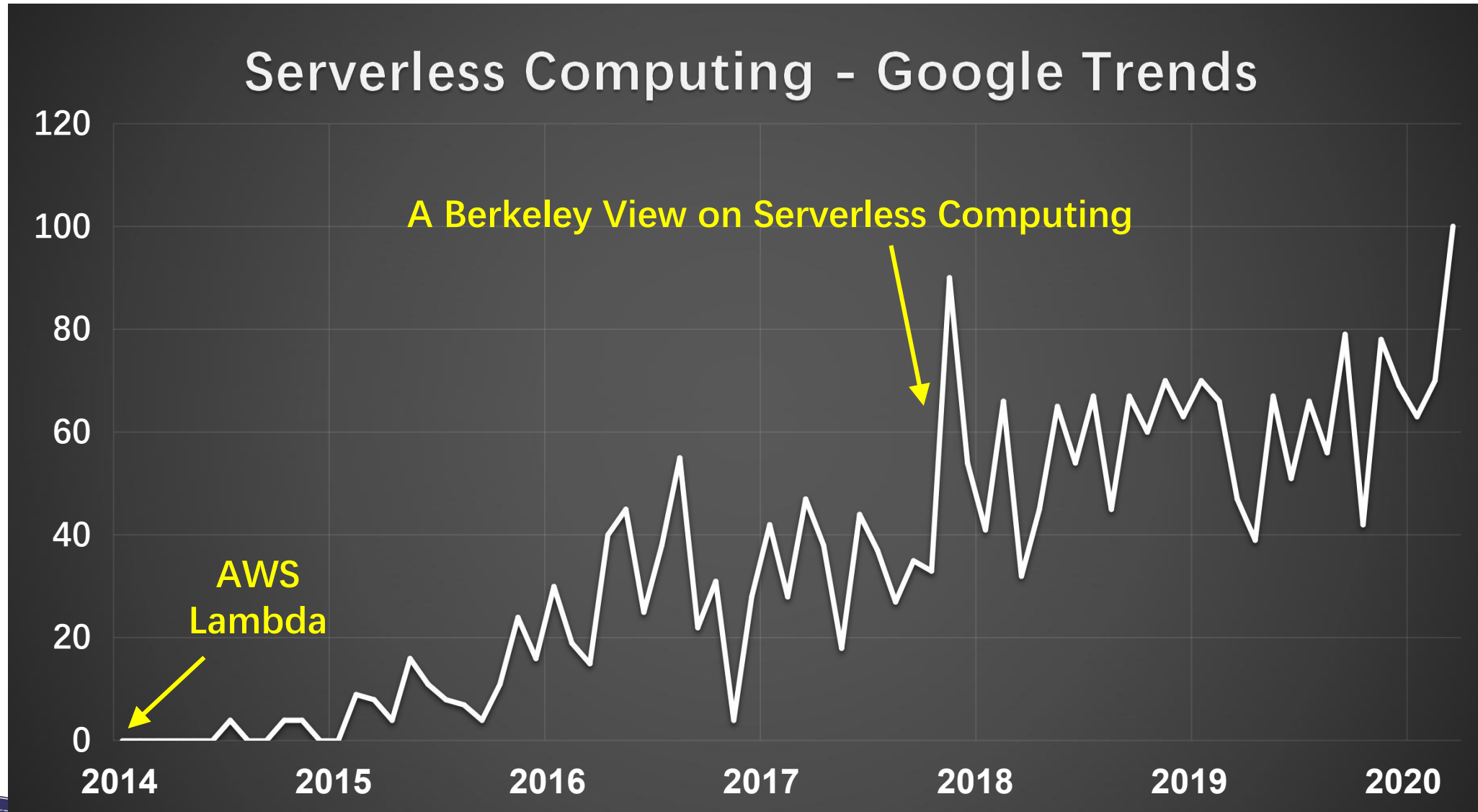


# INFless: A Native Serverless System for Low-Latency, High-Throughput Inference

Yanan Yang<sup>1</sup>, Laiping Zhao<sup>1</sup>, Yiming Li<sup>1</sup>, Huanyu Zhang<sup>1</sup>, Jie Li<sup>1</sup>,  
Mingyang Zhao<sup>1</sup>, Xingzhen Chen<sup>2</sup> and Keqiu Li<sup>1</sup>

Tianjin University<sup>1</sup>, 58.com<sup>2</sup>

# Background



# Background

- Existing Serverless Platforms
  - Commercial Services
    - AWS Lambda, Google Cloud Functions, Azure Functions, etc.
  - Open Source Communities
    - IBM OpenWhisk, OpenFaaS, Kubeless, Fission, etc.

Frameworks	OpenFaaS	OpenWhisk	Kubeless	Fission	Fn
Implementation	Go	Scala	Go	Go	Go
Language Support	Java, PHP, Python, Go, Node.js, etc.	JS, Swift, Python, PHP	Python, Ruby, Node.js, PHP	Go, .NET, Node.js, PHP, etc.	Go, Java, Node.js, Ruby, etc.
Function Workflow	---	Action Sequence	---	Fission Workflow	Fn Flow
Environment	Kubernetes Docker Swarm	Kubernetes Docker	Kubernetes	Kubernetes	Kubernetes Docker Swarm
Runtime	Container				








# Background

- Machine Learning (ML) Inference
  - China's largest local life service website
    - Deployed 600+ ML inference models
    - Serving millions of requests every minute
    - Latency critical with complex computations
    - More than 90% of them respond within 200ms
    - 400-server hybrid CPU/GPU cluster for inference



[www.58.com](http://www.58.com)

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

Service latency distribution



# Background

---

- Serving ML Inference on Serverless Systems
  - Advantages of serverless inference
    - Inference services could easily decouple from front-end applications (**stateless functions**)
    - Quickly deployment without participating in instance management (**function templates**)
    - The **auto-scaling ability** for varied workloads
    - The **pay-per-use billing model** saves much cost

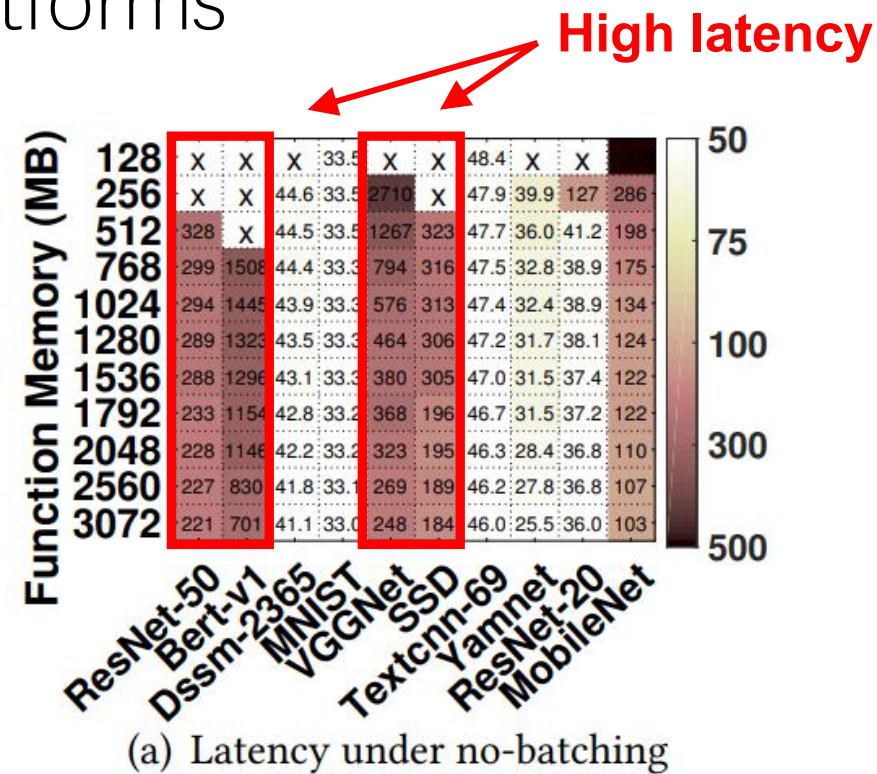


# Motivation

- Limitations of Existing Serverless Platforms

## 1 High Latency

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



- Small models (e.g., MNIST, Textcnn-69) can respond within 50ms.
- Even configured with the maximum function memory size, the execution time of some large models exceeds 200ms.



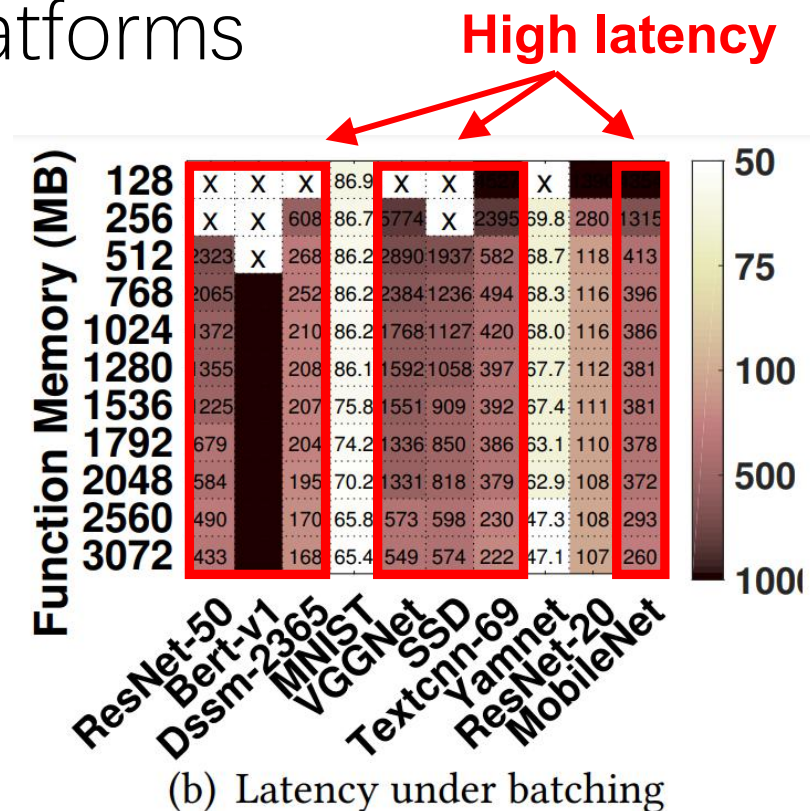


# Motivation

- Limitations of Existing Serverless Platforms

## 2 High Latency (batched request)

- 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.
- For batched request, nearly all inference models experience poor performance (latency increases more than 4x).

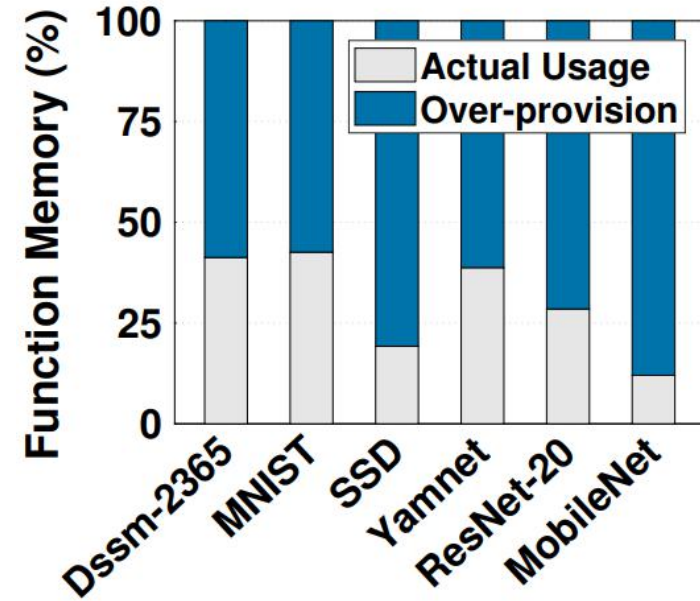


# Motivation

- Limitations of Existing Serverless Platforms

## 3 Resource Over-provisioning

- The proportional CPU-memory allocation policy set by a commercial serverless platform **encourages over-provisioning of memory**, resulting in poor resource utilization



(c) Function memory over-provisioning

- Commercial serverless providers only CPU-memory function resources.
- More than **50% of the function memory is over-provisioned** for serving these models to meet the latency SLO.



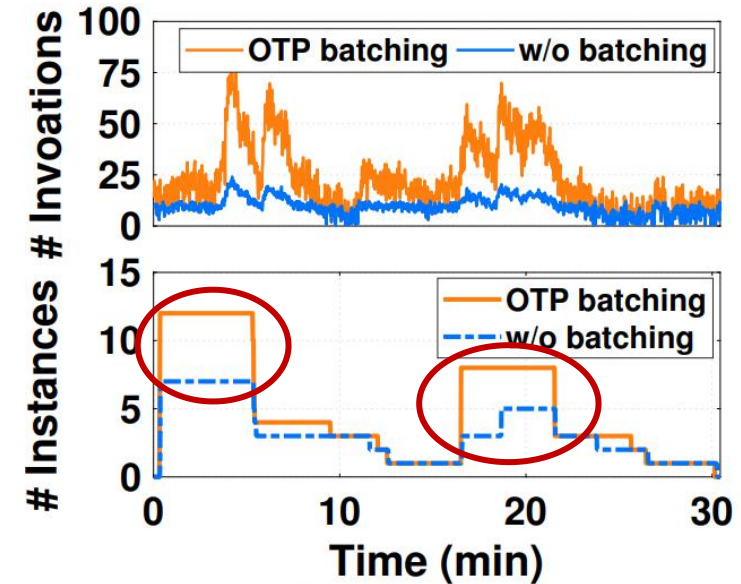


# Motivation

- Limitations of Existing Serverless Platforms

## 4 Low Throughput

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



(a) Excessive function scaling

- The "one-to-one mapping" policy inherently causes an excessive number of instances to be created, especially under bursty workloads.
- Batching could help to reduce much of the excessive instances.

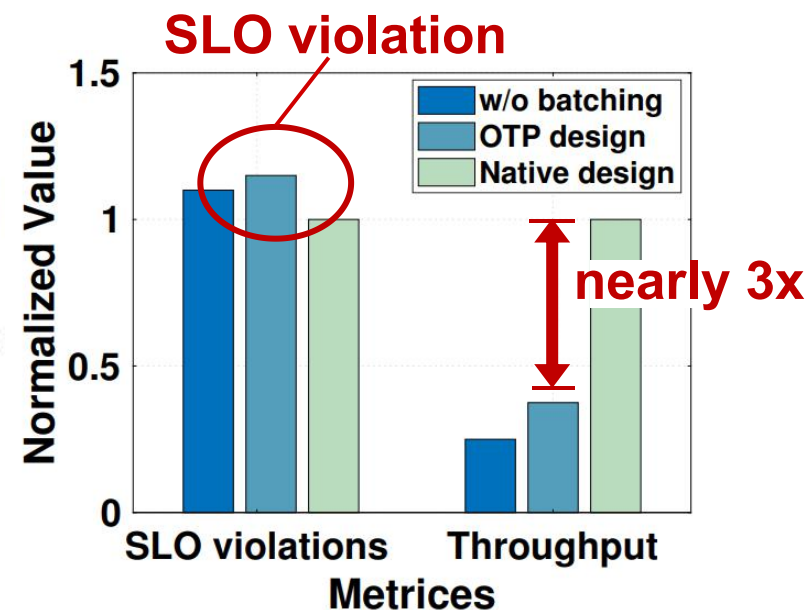


# Motivation

- Limitations of Existing Serverless Platforms

## 5 Shortages of OTP Batching

- OTP batching lacks the codesign of batch configuration, instance scheduling and resource allocation, **bringing only limited throughput improvement**



(b) OTP batching v.s. INFless

- On-Top-of-Platform (OTP) design, i.e., building another new buffer layer on top of the commercial serverless platform.
- The two layer designs only **have limited ability to optimize system throughput.**



# Motivation

- Implications
  - High latency
    - Observations #1 and #2
  - Resource-overprovisioning
    - Observations #3
  - Low throughput
    - Observations #4
  - Inefficient OTP design
    - Observations #5



Supporting hybrid  
CPU/accelerators



Producing resource-  
efficient scheduling



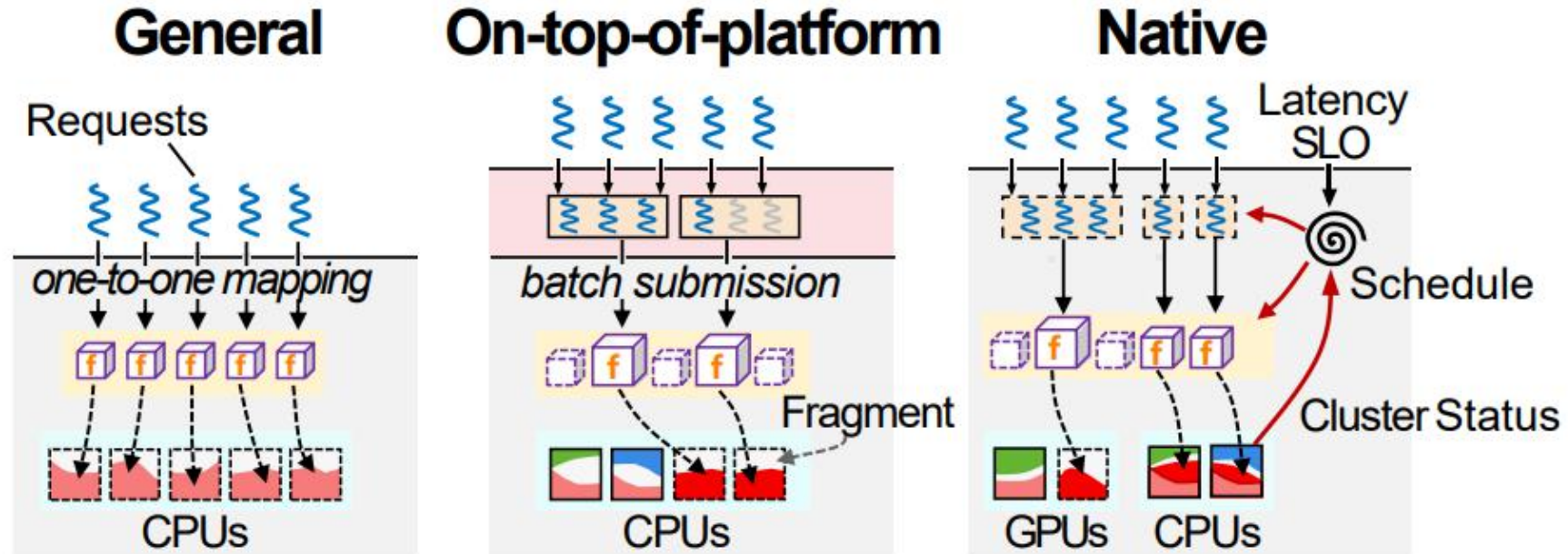
Function profiles

Supporting built-in  
batching



# Motivation

- Native Serverless Inference System



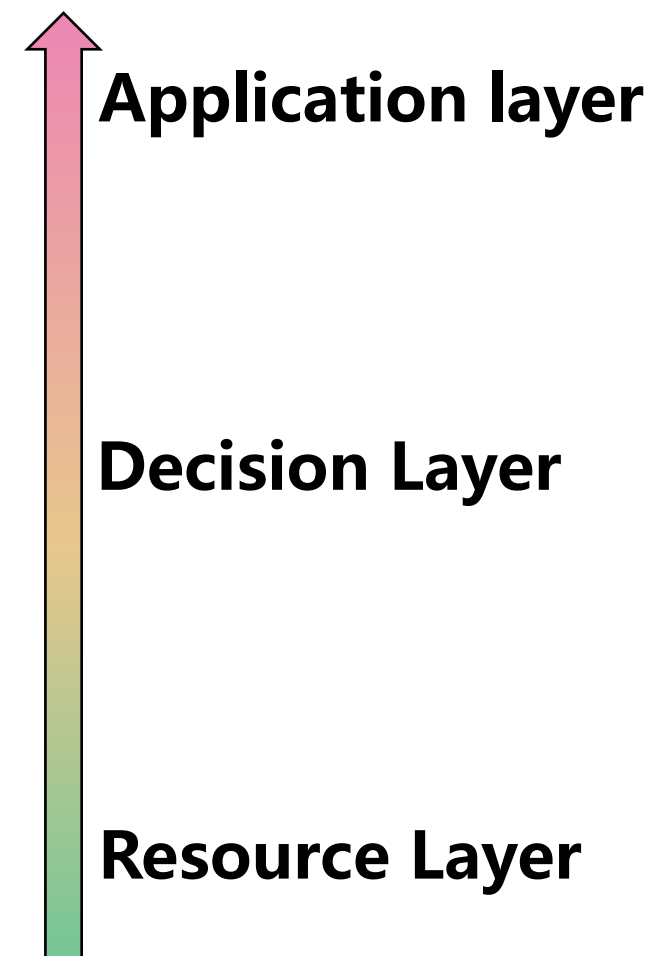
- A serverless inference platform should **exploit the features of inference** (e.g., shared operators, batching and computation intensive) to optimize system performance



# Motivation

---

- Challenges
  - **System running overhead**
    - Function profiling cost
    - Request forwarding, decision making, scheduling time, etc.
  - **Optimal scheduling decisions**
    - Batchsize, hardware selection, resource quotas
    - Instance placement
    - Workload dispatching rate
  - **Hybrid CPU/Accelerator provision**
    - Device collaboration
    - Hardware affinity & interchangeability
    - Prices

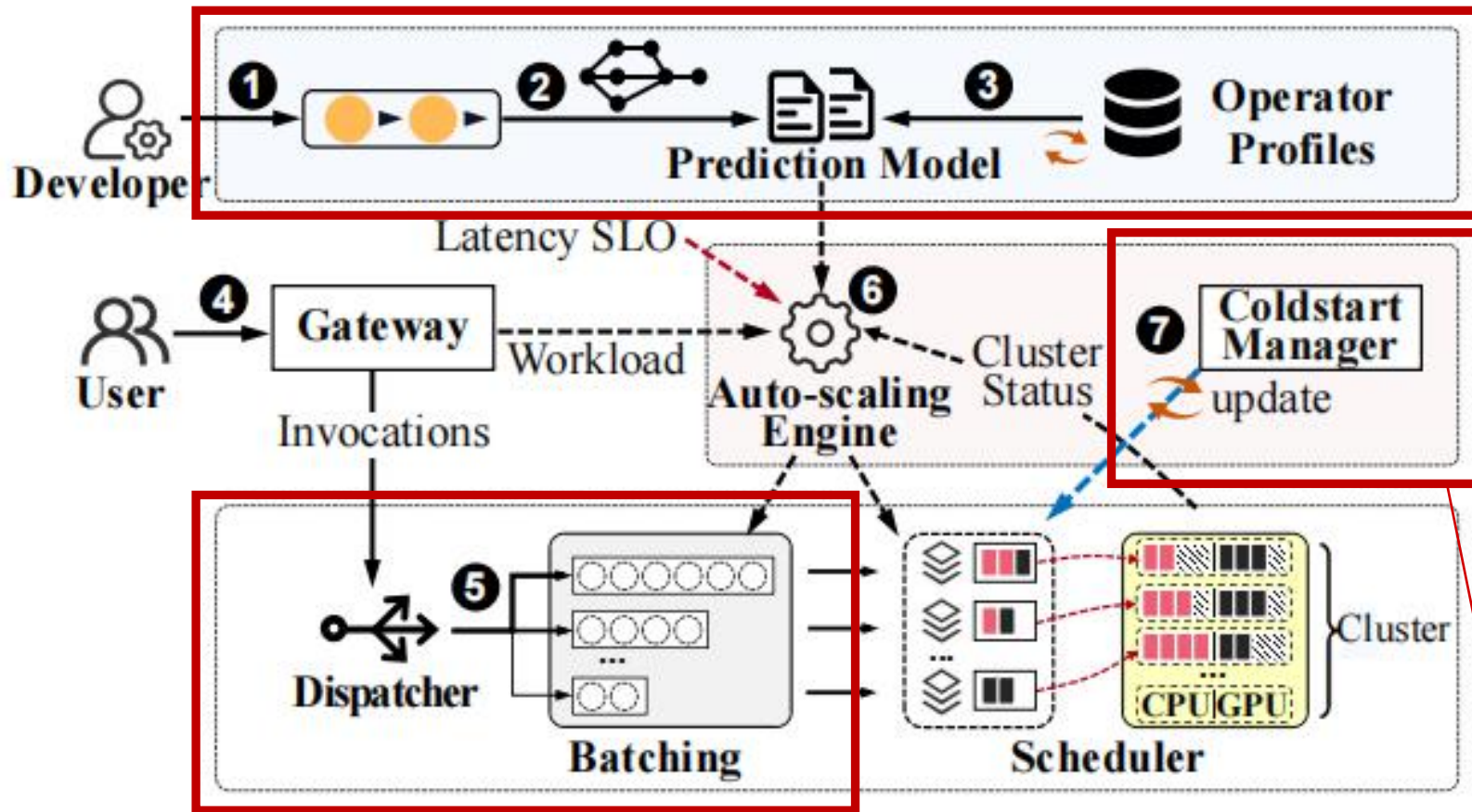




# INFless Design

- A domain-specific serverless platform for ML inference

Function profiling



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

Built-in, Non-uniform batching

Coldstart manager





# INFless Design

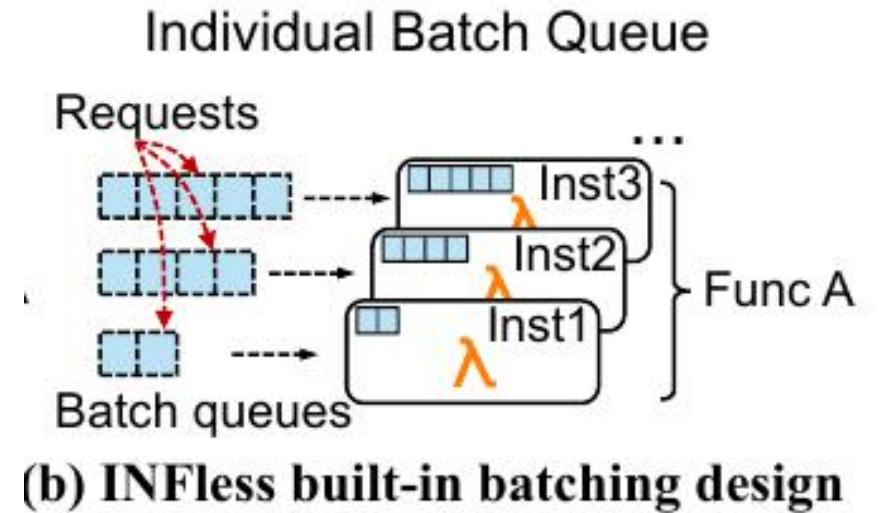
- Built-in, Non-uniform batching

- OTP batching

- Using a new buffer layer
    - Uniform scaling policy

- Built-in batching

- Integrated batch queues design
    - Non-uniform scaling policy

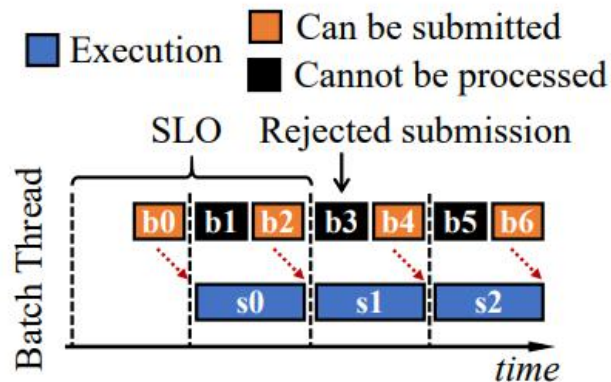


- **Built-in:** Batching is **integrated into** the serverless platform.
  - **Non-uniform:** Each instance has an **individual batch queue**. Instances of the same function may **have different configurations**.

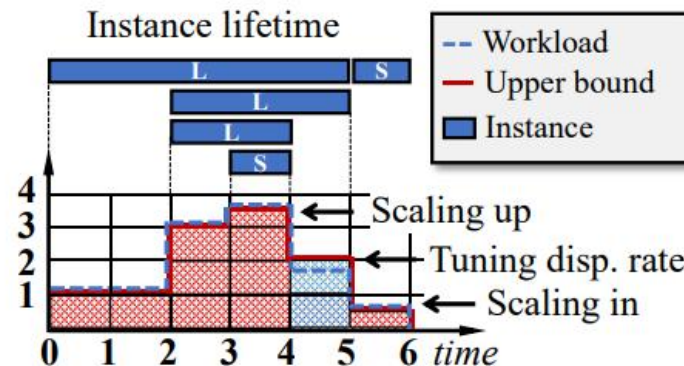


# INFless Design

- Built-in, Non-uniform batching

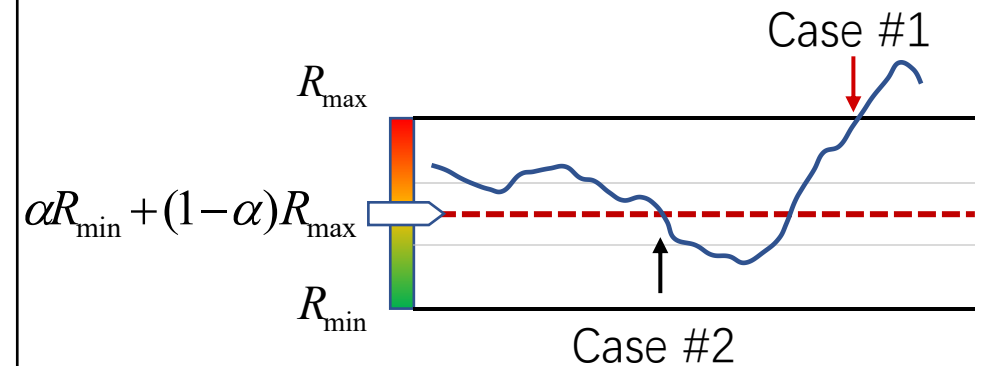


(a) Over-submission



(b) Instance scaling

- Scaling policy
  - Case 1: Scaling up/out
  - Case 2: Scaling down/in



- Batch-aware workload dispatcher

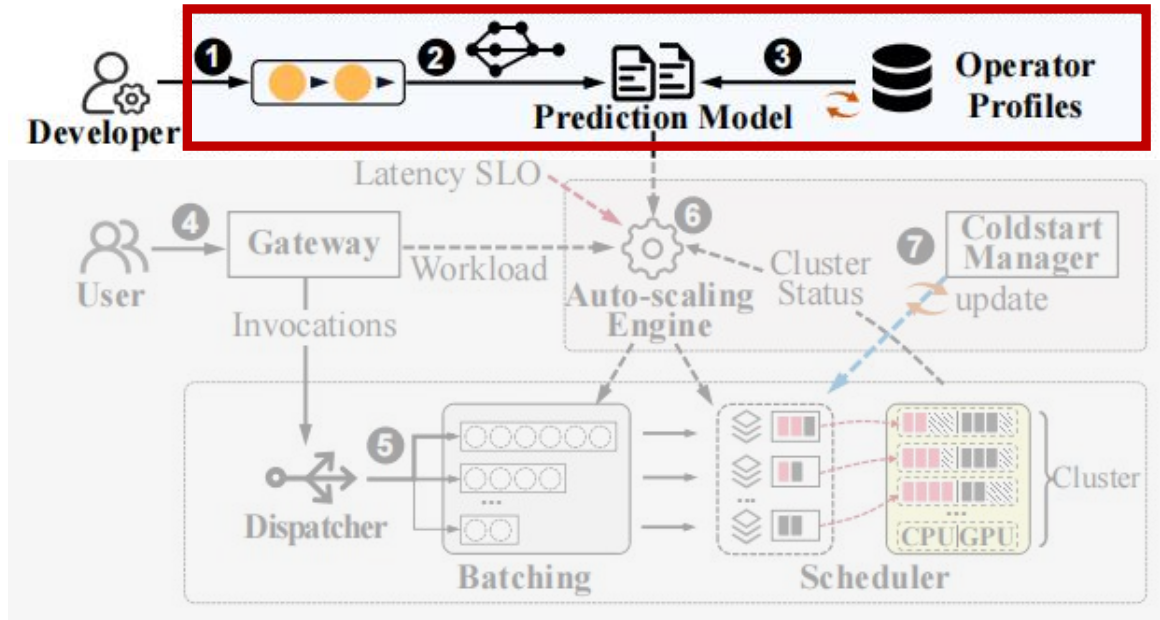
- The request arrival rate is strictly kept within a range of  $[r_{low}, r_{up}]$
- The lower (or upper) bound of workload can be processed in on instance

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



# INFless Design

- Combined Operator Profiling



- Inference function profiling
  - In 58.com, hundreds of inference models are deployed or updated every day
  - Offline profiling every inference function is rather costly

- How could we reduce the inference function profiling cost?

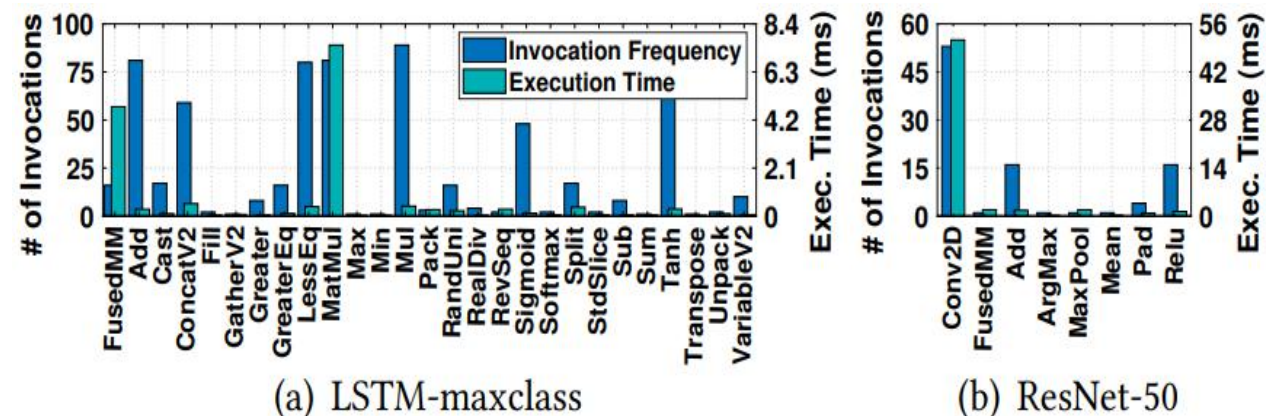


# INFless Design

- Combined Operator Profiling
  - Model structure
    - These models has 1000+ calls of operators
    - Total 71 distinct operators

**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]



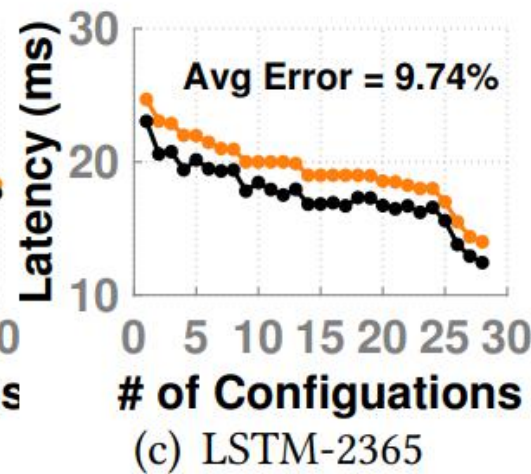
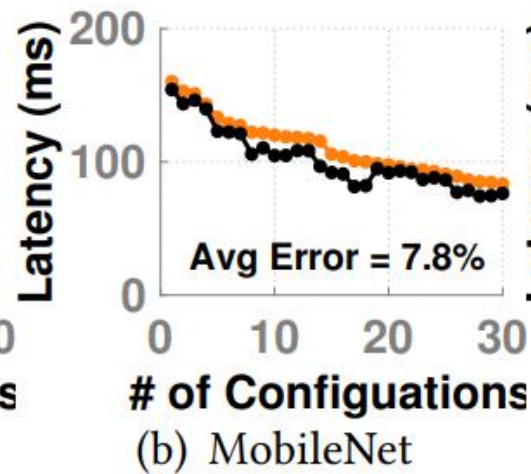
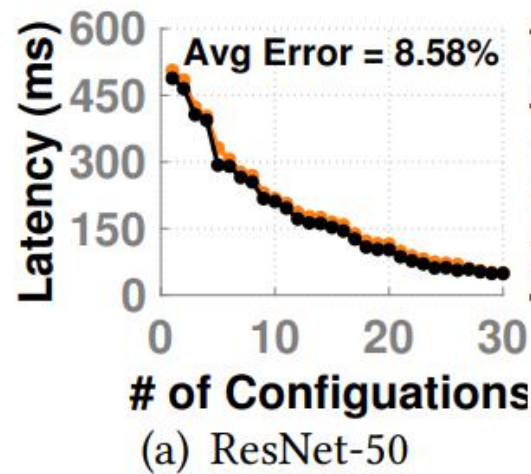
- Inference functions **share a common set of operators**, and the execution time is dominated by a small subset of them.





# INFless Design

- Combined Operator Profiling
  - Operator-level profiling
    - 5 tuple profiles:  $o_i = (p_i, b_i, c_i, g_i, t_i)$   
<input size, batchsize, CPU quotas, GPU quotas, execution time>
    - DNN task graph  $G = (O, E)$

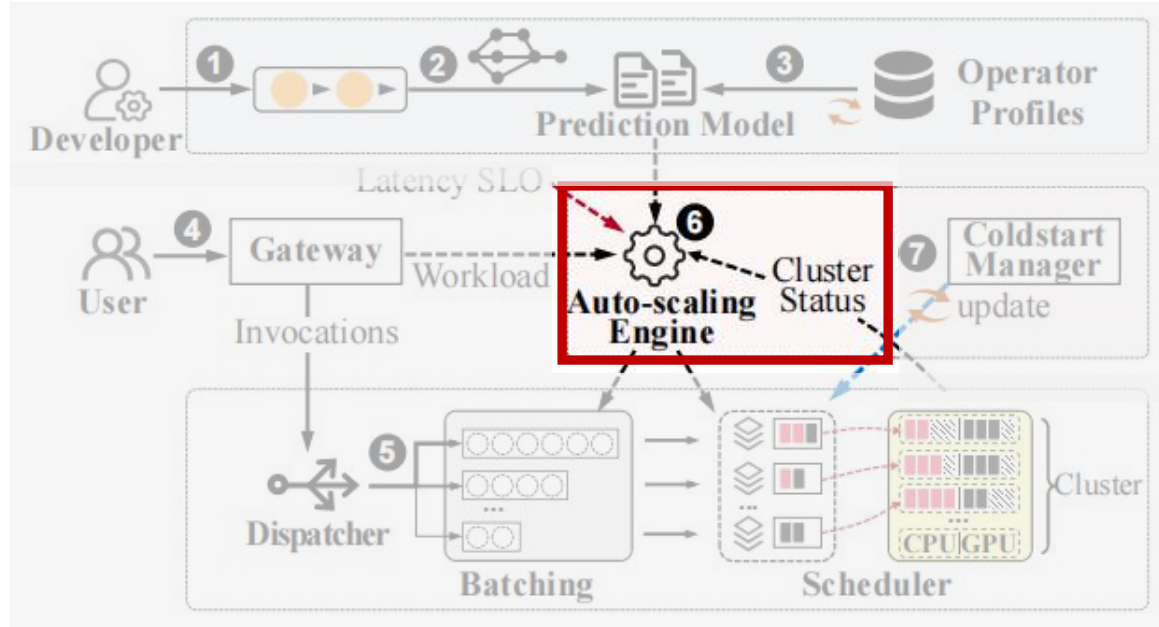


- Sequence chain  
$$t_{chain} = \sum_{i \in \{chain\}} t_i$$
- Parallel branches  
$$t_{branches} = \max_{i \in \{branches\}} t_i$$



# INFless Design

- Resource Efficient Scheduling



- Input
  - function profiles
  - workload (req/s)
  - cluster status (server usage)
- Output
  - number of instances
  - batchsize
  - CPU/GPU quotas
  - placement





# INFless Design

- How to guarantee the end-to-end latency SLO?

- Latency breakdown

- Batch queuing time
    - Batch execution time
    - Cold start time

$$l = t_{batch} + t_{exec} + miss_{rate} \cdot t_{cold}$$

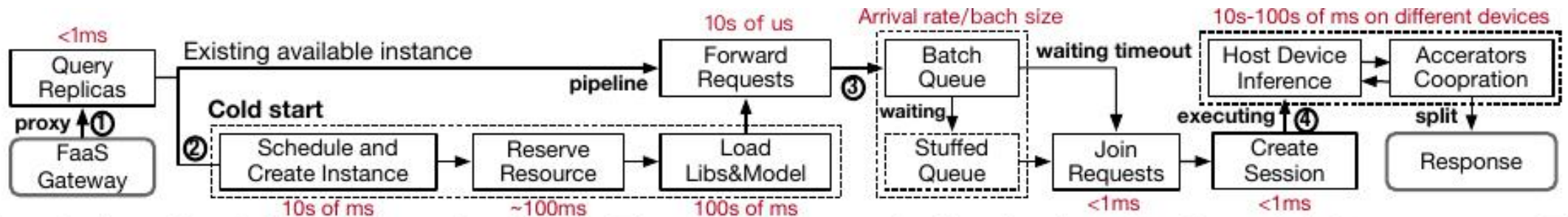


Figure 1: Serverless inference latency breakdown. The red numbers are the duration time of each step in inference process. The **proxy** and **pipeline** path are the process of receiving a request and forwarding to the service end-point in Gateway, and **join** and **split** are the fusion and parsing of requests for batching process, respectively.



# INFless Design

- Resource Efficient Scheduling
  - Problem formalization

$$\text{minimize : } \sum_{j=1}^m (\beta C_j + G_j) y_j \quad (2)$$

$$t_{wait}^i + t_{exec}^i \leq t_{slo}^i, \quad \forall i \in [1, \dots, n] \quad (3)$$

$$t_{exec}^i \leq t_{wait}^i, \quad \forall i \in [1, \dots, n] \quad (4)$$

$$\sum_i^n c_i x_{ij} \leq C_j y_j, \quad \forall j \in [1, \dots, m] \quad (5)$$

$$\sum_i^n g_i x_{ij} \leq G_j y_j, \quad \forall j \in [1, \dots, m] \quad (6)$$

$$\alpha R_{max}^k + (1 - \alpha) R_{min}^k \leq R_k \leq R_{max}^k, \quad \forall k \in I \quad (7)$$

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

$$b_i, c_i \in \mathbb{Z}_+, \quad g_i \in \mathbb{Z} \quad (9)$$

← • SLO constraint

← • Resource capacity constraint

← • Workload constraint

← • Variables

NP-hard bin packing problem



# INFless Design

- Resource Efficient Scheduling
  - Greedy scheduling algorithm

```

1   $n_k \leftarrow 0, x_{ij} \leftarrow 0;$ 
2  while  $R_k > 0$  do
3      for  $b \in \mathbf{B}$  do
4           $I_b \leftarrow \text{AvailableConfig}(b, R_k, t_{slo})$ 
5          /* e.g.,  $I_b = \{\langle b, c_1, g_1 \rangle, \dots, \langle b, c_n, g_n \rangle\}$  */
6          if  $I_b = \text{NULL}$  then
7              continue; // try next largest batchsize
8          for  $\forall \langle b, c_i, g_i \rangle \in I_b, \forall j \in \mathbf{M}$  do
9              Derive the res. efficiency  $e_{ij}$  using Equation 10;
10
11          if  $\{e_{ij}\} \neq \text{NULL}$  then
12               $i, j \leftarrow \text{argmax}\{e_{ij}\}, \forall i \in [1..n], j \in [1..m];$ 
13               $n_k \leftarrow n_k + 1, x_{ij} \leftarrow 1;$  // schedule  $i$  to  $j$ 
14               $\langle C_j, G_j \rangle \leftarrow \langle C_j, G_j \rangle - \langle c_i, g_i \rangle;$ 
15               $R_k \leftarrow R_k - r_{up};$  // update  $R_k$ 
16          break; // scaling for the rest of  $R_k$ 

```

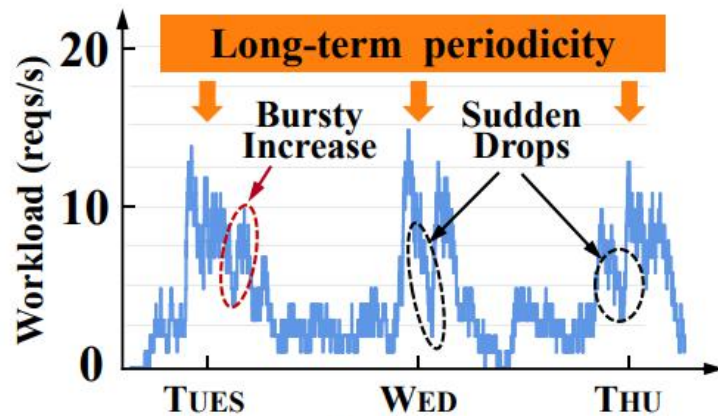
- Explore from the largest batchsize
  - get available configurations that could meet SLO
- Iterate server cluster
  - drive the resource efficiency
  - choose the maximum  $e_{ij}$
  - schedule instance
- Next loop for the residual workload

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

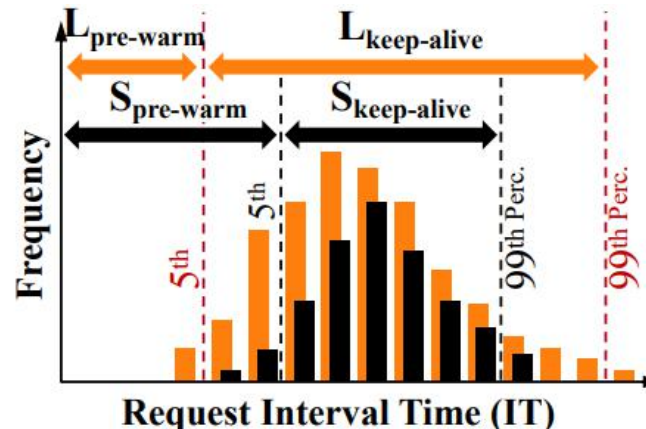


# INFless Design

- Long-short Term Hybrid Histogram Policy (LSTH)
  - Inference workload characterization
    - Long-term periodicity
    - Short-term burst
  - Hybrid histogram policy (HHP)
    - pre-warming window, keep-alive window



(a) Workload features



(b) Weighted hybrid hist policy

- Weighted method

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

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

$$\gamma = 0.5$$

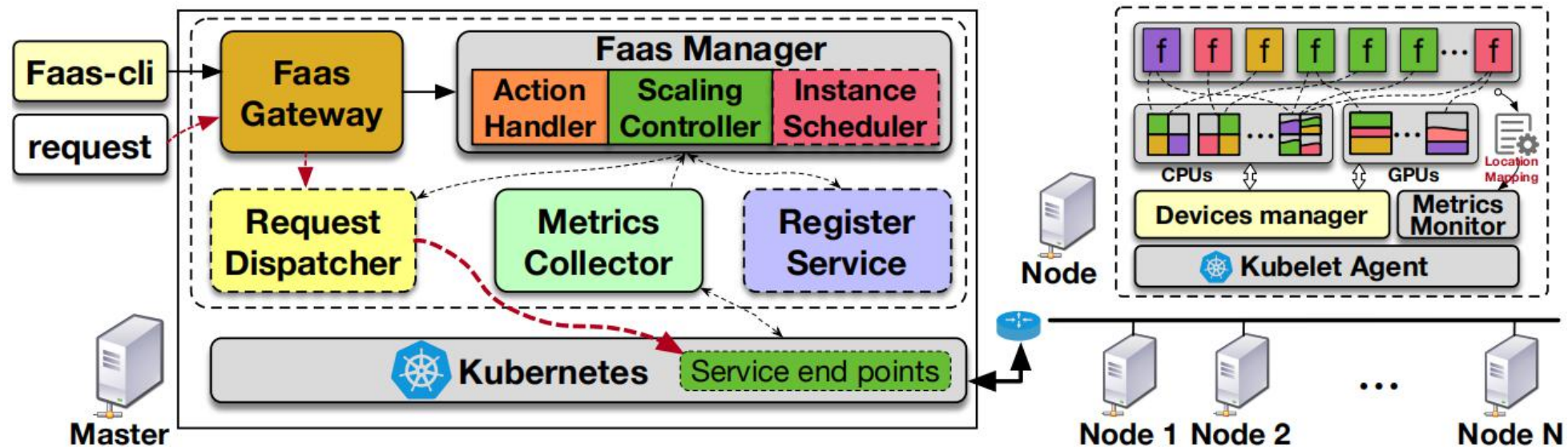




# INFless Design

- System Implementation

GitHub: <https://github.com/TankLabTJU/INFless>



- Developed based on OpenFaaS
- faas-cli, faas-gateway and faas-netes modules
- Nvidia MPS, Kubernetes cluster (Docker)
- 9,300+ LOCs (Golang)



# Evaluation

- Workloads:
  - Online secondhand vehicle trading (OSVT) business
  - Q&A robot service
  - 3 production traces from Azure Function [Shahrad 2020]
- Testbed:
  - local cluster experiments (8-server cluster)
  - large scale of simulations (2000-server cluster)

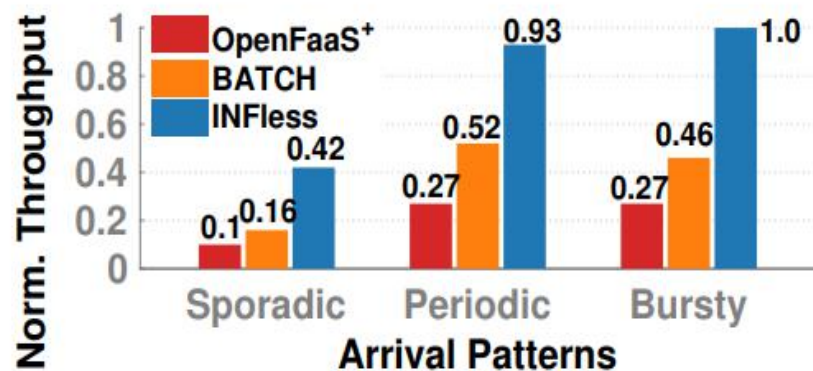
Component	Specification	Component	Specification
CPU device	Intel Xeon Silver-4215	Shared LLC Size	11MB
Number of sockets	2	Memory Capacity	128GB
Processor BaseFreq.	2.50 GHz	Operating System	Ubuntu 16.04
CPU Threads	32 (16 physical cores)	SSD Capacity	960GB
GPU device	Nvidia RTX 2080Ti	GPU Memory Config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



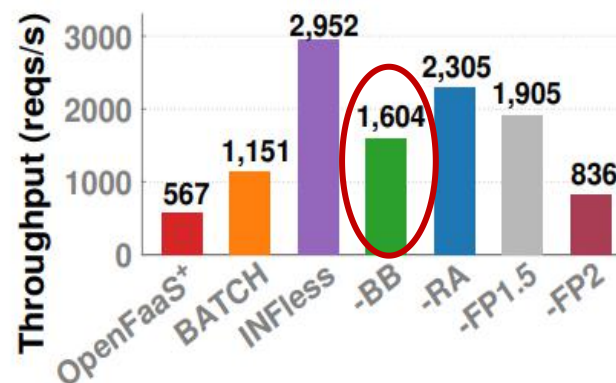


# Evaluation

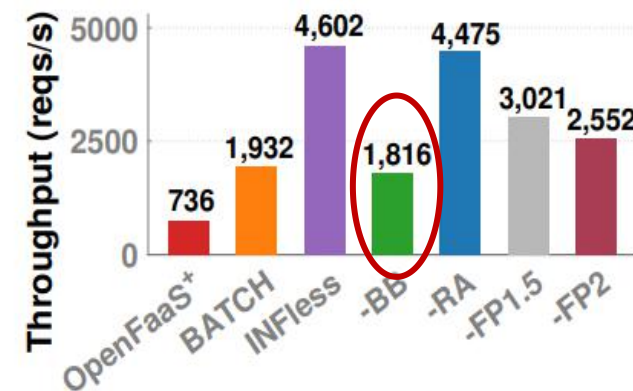
- High Throughput
  - INFless improves system throughput by  $5.2\times$  and  $2.6\times$  on average compared with baselines.
  - Every component of INFless contributes much to throughput improvement, with batching being highest.



(a) Thp. under production traces



(a) OSVT



(b) Q&A robot

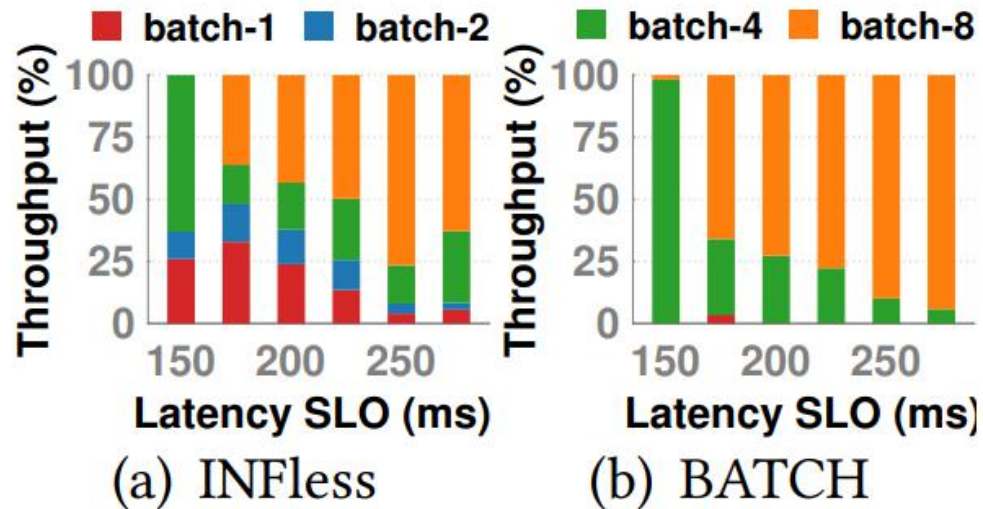
Throughput comparison

Component analysis of INFless



# Evaluation

- Less over-provisioning



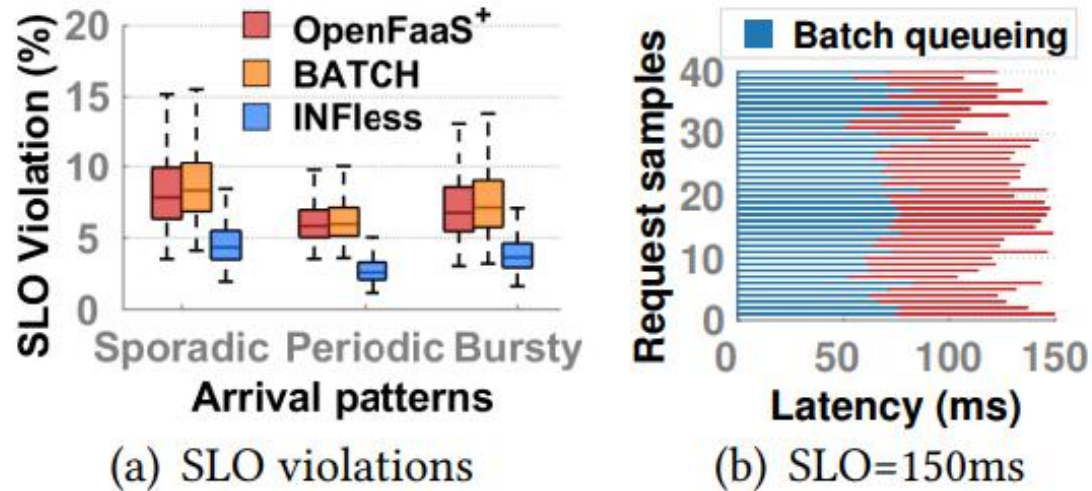
Throughput contribution from different batchsizes

- INFless opts for flexible configurations on both batchsizes and resource allocations.
- INFless can reduce the provisioned resources by 60% compared with BATCH.



# Evaluation

- SLO Violation



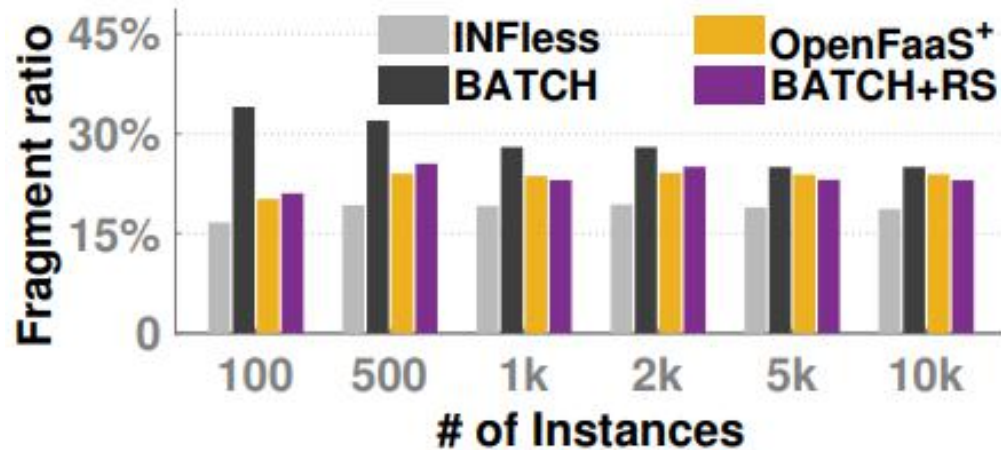
SLO violation rate and latency breakdown

- INFless can guarantee the latency SLO of inference workloads.
- The SLO violation rate by INFless is  $\leq 3.1\%$  on average, which is far lower than that of OpenFaaS+ and BATCH.



# Evaluation

- Resource Fragments



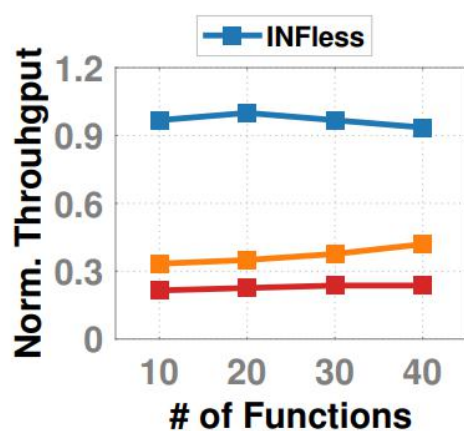
Resource fragments in large-scale simulations

- INFless's resource-aware scheduling algorithm reduces the resource fragments significantly.
- INFless generates a resource fragment ratio as low as 15%, which is much lower than the other systems.

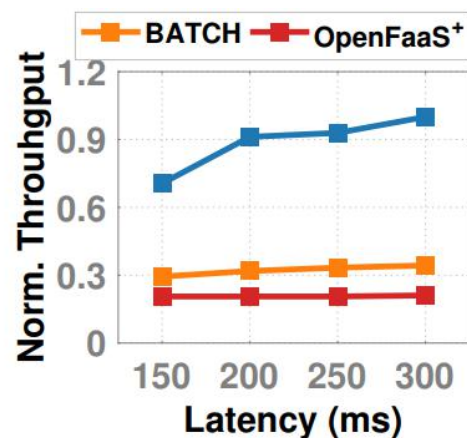


# Evaluation

- Scalability & Cost-efficiency



(a) Thp. under different # of funcs



(b) Thp. under different SLOs

**Table 4: Computation cost comparison.**

	AWS EC2	OpenFaaS <sup>+</sup>	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 [\$]	$2.23 \times 10^{-5}$	$2 \times 10^{-5}$	$1.32 \times 10^{-5}$	$1.6 \times 10^{-6}$

- When the concurrent requests scales to 10,000, INFless still achieves 2.6× and 4.2× higher throughput than BATCH and OpenFaaS+.

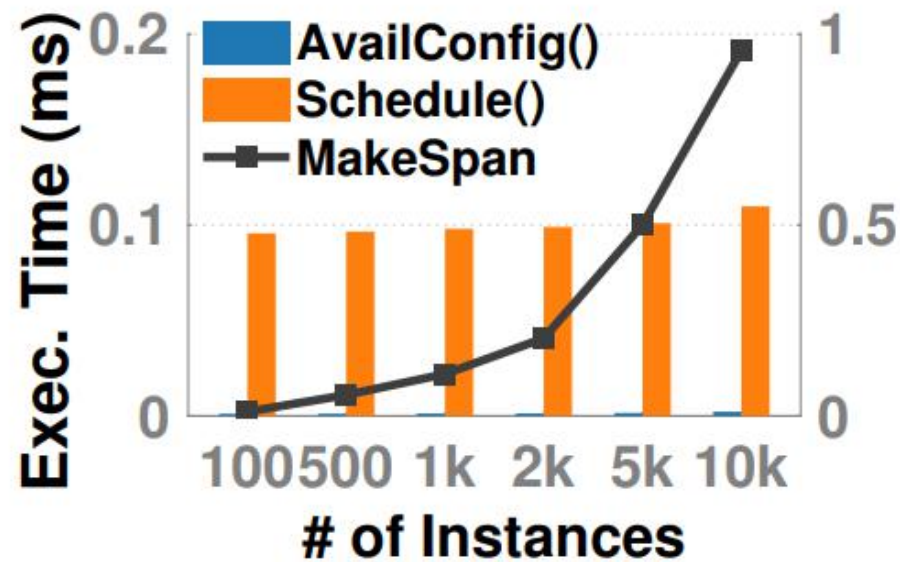
- Considering the 400-server production cluster in 58.com with 1.9 billion requests per day, INFless could save about \$1,200,000 every year.





# Evaluation

- **Scheduling Overhead**



Scheduling latency in large-scale simulations

- INFless scales well in large-scale evaluations
- 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**





# Conclusion

---

- Serverless computing has been a success in multiple areas, ML inference **is yet another promising area** for serverless adoption.
- Achieving both **low latency** and **high resource efficiency** in current serverless platforms is challenging.
- A **domain-specific** serverless system design for inference could help to address this problem, which also **requires full-stack optimizations** across both software and hardware layers.
- **INFless is just a start!**



---

# Thank you

