上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

Serverless Application in Machine Learning

马汝辉 副教授 博导
计算机科学与工程系
上海交通大学

饮 水 思 源 • 爱 国 荣 校

目录

01

NumPyWren

- Serverless Linear Algebra

NumPyWren Overview

Background

- Current distributed programming abstractions such as MPI and MapReduce rely on the tightly integrated resources in a collection of individual servers.
- To write applications for a disaggrated datacenter, the datacenter operator must expose a new programming abstraction.

Motivation

- Serverless computing is a programming model in which the cloud provider manages the servers, and also dynamically manages the allocation of resources.
- Disaggregation can provide benefits to linear algebra tasks as these workloads have large dynamic range in memory and computation requirements.

Contribution

- large scale linear algebra algorithms can be efficiently executed using **stateless functions and disaggregated storage**
- design LAmbdaPACK, a **domain specific language for linear algebra algorithms**
- NumPyWren can **scale** to run Cholesky decomposition

Background: Serverless Computing

Cloud providers offer the ability to execute functions on demand, hiding cluster configuration and management overheads from end users.

① **Cloud providers offer a number of storage options ranging from key-value stores to relational databases.**

- The cost of data storage in an object storage system is often orders of magnitude lower when compared to instance memory.

② **Cloud providers also offer publish-subscribe services like Amazon SQS or Google Task Queue.**

Amazon S3

amazon SQS

Background: Serverless Computing

Cloud providers offer the ability to execute functions on demand, hiding cluster configuration and management overheads from end users.

③ **Computation resources offered in serverless platforms are typically restricted to a single CPU core and a short window of computation.**

- AWS Lambda provides 900 seconds of compute on a single AVX core with access to up to 3 GB of memory and 512 MB of disk storage.

④ **The linear scalability in function execution is only useful for embarrassingly parallel computations when there is no communication between the individual workers.**

Background: Linear Algebra Algorithms

- Cholesky factorization is one of the most popular algorithms for solving linear equations, and it is widely used in applications such as matrix inversion, partial differential equations, and Monte Carlo simulations.

$Ax = b$	
$A = LL^T$	$O(n^3)$
$Ly = b$	$O(n^2)$
$L^T x = y$	$O(n^2)$

Communication-Avoiding Cholesky

Algorithm 1 Communication-Avoiding Cholesky [5]

Input:

A - Positive Semidefinite Symmetric Matrix

B - block size

N - number of rows in A

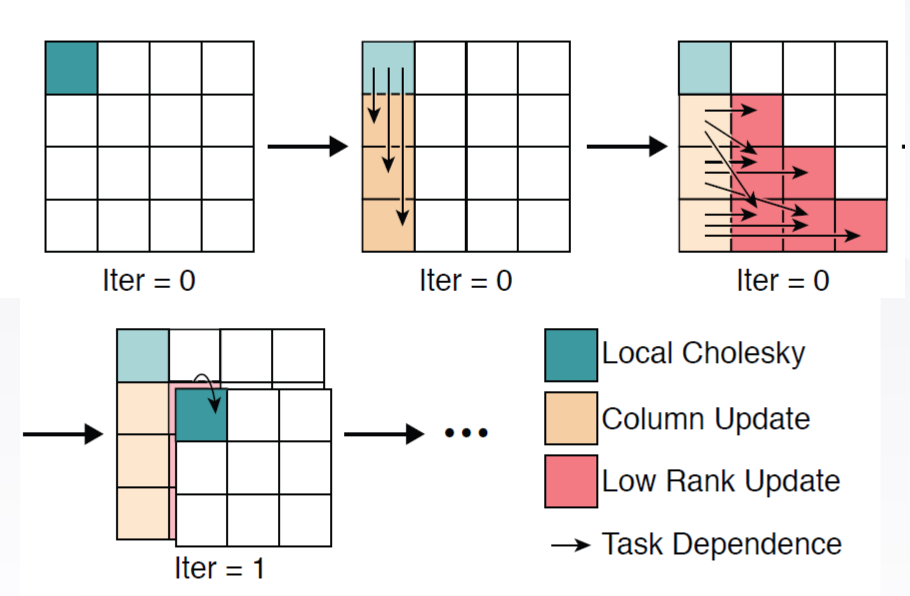
Blocking:

A_{ij} - the ij-th block of A

Output:

L - Cholesky Decomposition of A

for $j \in \{0 \ldots \lceil \frac{N}{B} \rceil\}$ **do**
  $L_{jj} \Leftarrow cholesky(A_{jj})$
  for all $i \in \{j+1 \ldots \lceil \frac{N}{B} \rceil\}$ **do in parallel**
    $L_{ij} \Leftarrow L_{jj}^{-1} A_{ij}$
  end for
  for all $k \in \{j+1 \ldots \lceil \frac{N}{B} \rceil\}$ **do in parallel**
    for all $l \in \{k \ldots \lceil \frac{N}{B} \rceil\}$ **do in parallel**
      $A_{kl} \Leftarrow A_{kl} - L_{kj}^T L_{lj}$
    end for
  end for
end for



① **Diagonal block Cholesky decomposition**

② **Parallel column update**

③ **Parallel submatrix update**

④ **Diagonal block Cholesky decomposition**

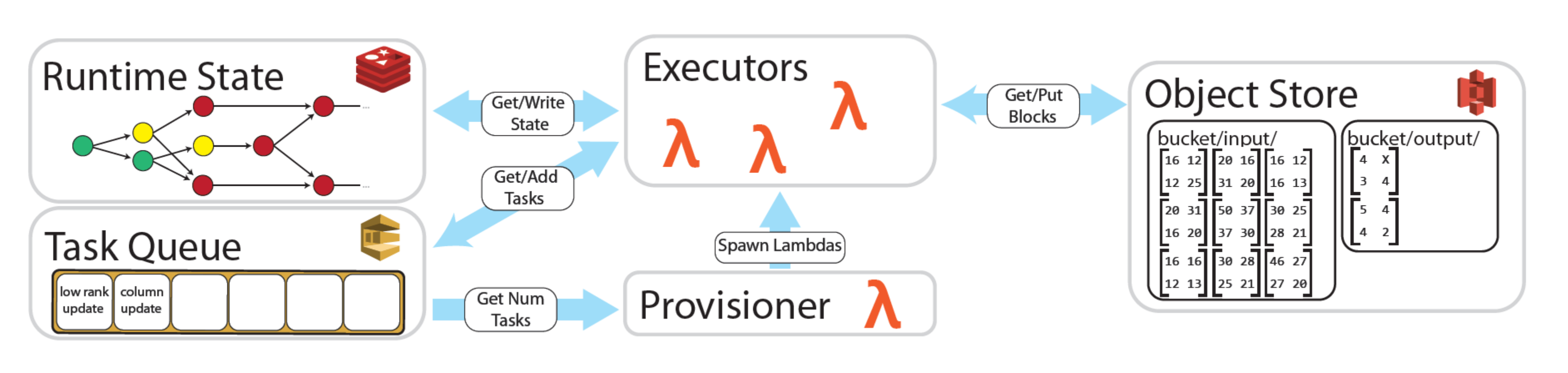
❶ dynamic parallelism

❷ fine-grained dependencies

SJTU

System Design

- **Task Enqueue: enqueue the first task that needs to be executed into the task queue**
- **Executor Provisioning: launch an executor, and maintain the number of active executors based on task queue size**
- **Task Execution: manage executing and scheduling NumPyWren tasks**
- **Runtime State Update: update the task status in the runtime state store**



The architecture of the execution framework of NumPyWren showing the runtime state during a 6x6 Cholesky decomposition. The first block Cholesky instruction has been executed as well as a single column update.

System Design

Fault tolerance in NumPyWren is much simpler to achieve due to the disaggregation of compute and storage.

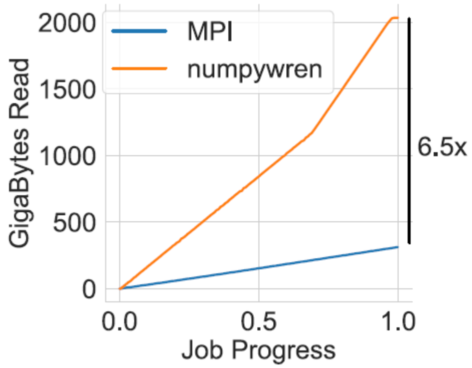
- **Task Lease: NumPyWren executes failed tasks via a lease mechanism, which allows the system to track task status without a scheduler periodically communicating with executors.**
- **Failure Detection and Recovery: Failure detection happens through lease expiration and recovery latency is determined by lease length.**
- **Garbage Collection: it is imperative we clear the state when it is no longer necessary.**
- **Autoscaling**
  - Task scheduling and worker management is decoupled in NumPyWren, which allows auto-scaling of computing resources for a better cost-performance trade-off.
  - We adopt a simple auto-scaling heuristic and it achieves good utilization while keeping job completion time low.

Evaluation

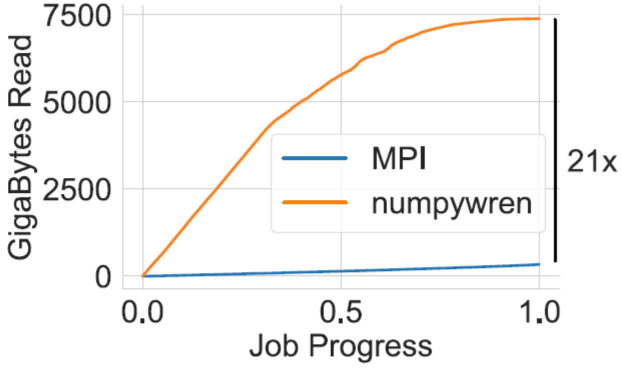
System Comparisons

- The amount of bytes read by NumPyWren is always greater than MPI.
- Even though NumPyWren reads more than 21x bytes over the network when compared to MPI, our end to end completion time is only 47% slower.

Algorithm	MPI (sec)	NumPyWren (sec)	Slow down
SVD	5.8e4	4.8e4	N/A
QR	9.9e3	1.4e4	1.5x
GEMM	5.0e3	8.1e3	1.6x
Cholesky	1.7e3	2.5e3	1.5x



(a) GEMM



(b) QR

Evaluation

System Comparisons

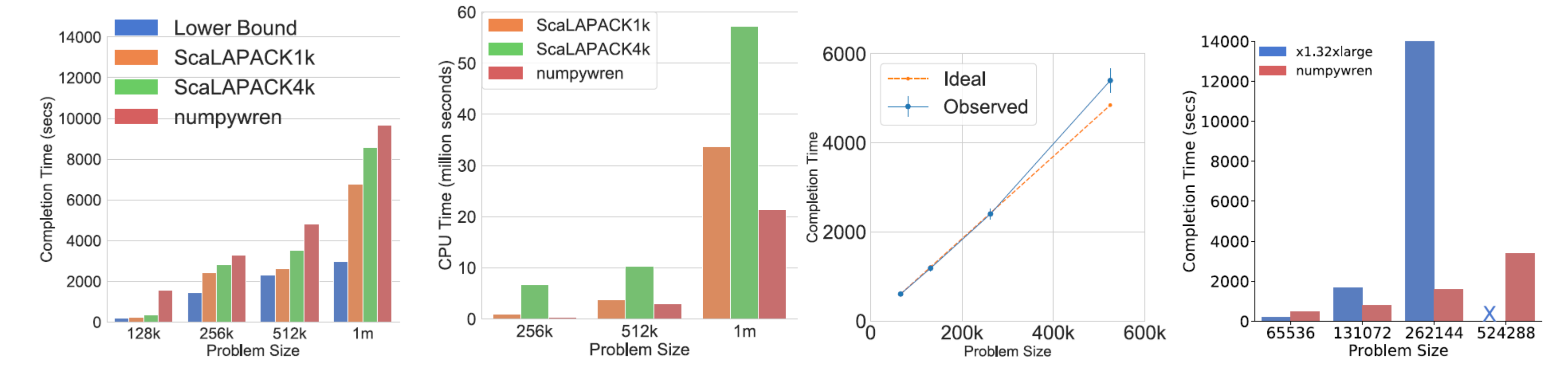
- For MPI the core-seconds is the total amount of cores multiplied by the wall clock runtime.
- For NumPyWren we wish to only account for “active cores" in our core-second calculation, as the free cores can be utilized by other tasks.
- NumPyWren can achieve resource savings of over 3x for the SVD algorithm.

Algorithm	MPI (core-secs)	NumPyWren (core-secs)	Resource saving
SVD	2.1e7	6.2e6	3.4x
QR	2.6e6	2.2e6	1.15x
GEMM	1.2e6	1.9e6	0.63x
Cholesky	4.5e5	3.9e5	1.14x

Evaluation

Scalability

a) Completion time on various problem sizes when NumPyWren is run on same setup as ScaLAPACK
b) Total execution core-seconds for Cholesky when the NumPyWren and ScaLAPACK are optimized for utilization.
c) Weak scaling behavior of NumPyWren.
d) Comparison of NumPyWren with 128 core single node machine running Cholesky decompositions of various sizes



(a) (b) (c) (d)

02

Cirrus

- Cirrus: a Serverless Framework for End-to-end ML Workflows

Cirrus Overview

Background

- The widespread adoption of ML techniques in a wide-range of domains has made machine learning one of the leading revenue-generating datacenter workloads.
- The complexity of ML workflows leads to two problems, over-provisioning and explicit resource management.

Motivation

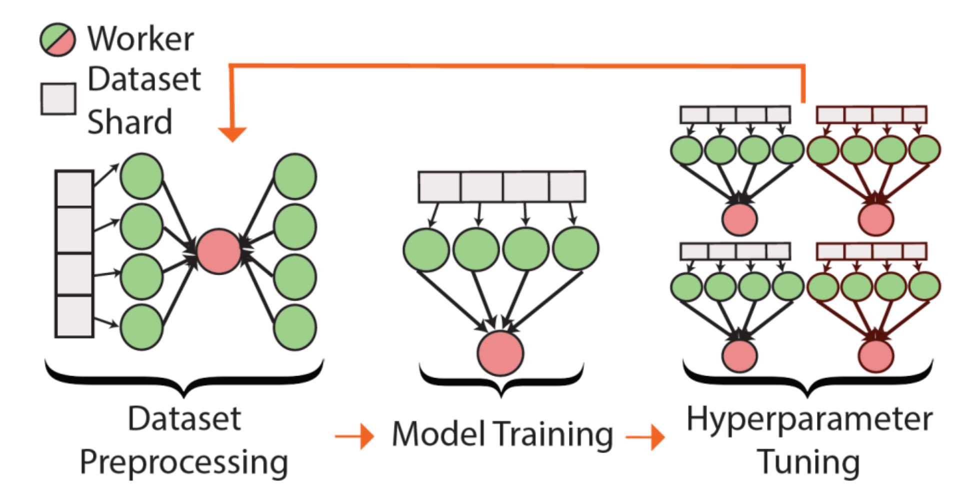
- Serverless computing relies on the cloud infrastructure to automatically address the challenges of resource provisioning and management.
- The benefits of serverless computing for ML hinge on the ability to run ML algorithms efficiently.

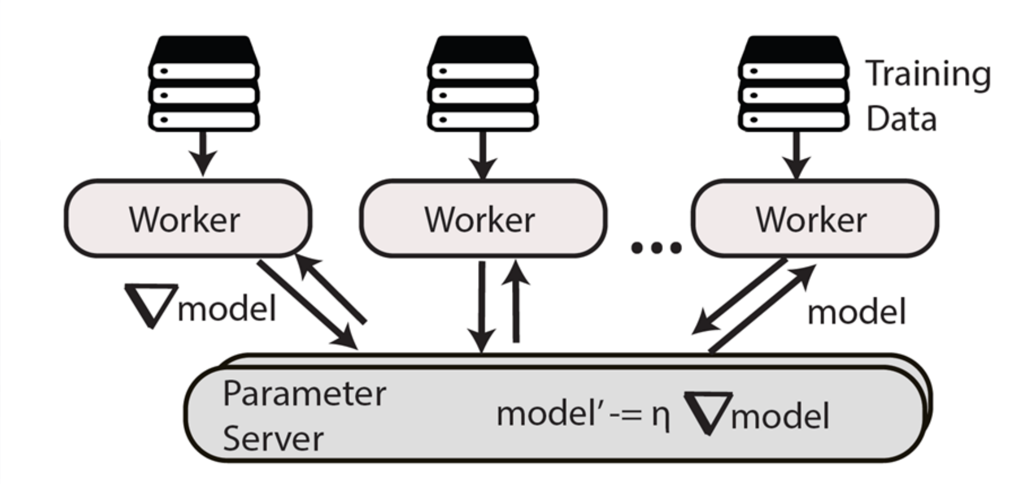
Contribution

- Cirrus is designed to efficiently support the entire ML workflow.
- Cirrus builds on three key design properties, ultra-lightweight, cost-saving, and stateless.
- It yields a 3.75x improvement on time-to-accuracy compared to the best-performing configuration ML specialized frameworks.

Background: End-to-end ML Workflow

- Dataset preprocessing typically involves an expensive map/reduce operation on data.
- Model training: Workers consume data shards, compute gradients, and synchronize with a parameter server.
- Hyperparameter optimization to tune model and training parameters involves running multiple training instances.





Background: Challenges

Machine Learning

- **Over-provisioning:** The heterogeneity of the different tasks in an ML workflow leads to a significant resource imbalance during the execution of a training workflow.
- **Explicit resource management:** Systems that leverage VMs for machine learning workloads generally require users to repeatedly perform a series of onerous tasks.

Serverless Computing

- **Small local memory and storage:** Lambda functions, by design, have very limited memory and local storage.
- **Low bandwidth and lack of P2P communication:** Lambda functions have limited available bandwidth when compared with a regular VM.
- **Short-lived and unpredictable launch times:** Lambda functions are short-lived and their launch times are highly variable.
- **Lack of fast shared storage:** Because lambda functions cannot connect between themselves, shared storage needs to be used.

Design: Principles

- **Adaptive, fine-grained resource allocation**
  - To avoid resource waste that arises from over-provisioning, Cirrus should flexibly adapt the amount of resources reserved for each workflow phase with fine-granularity.
- **Stateless server-side backend**
  - To ensure robust and efficient management of serverless compute resources, Cirrus, by design, operates a stateless, server-side backend.
- **End-to-end serverless API**
  - Model training is not the only important task an ML researcher has to perform.
- **High scalability**
  - ML tasks are highly compute intensive, and thus can take a long time to complete without efficient parallelization.

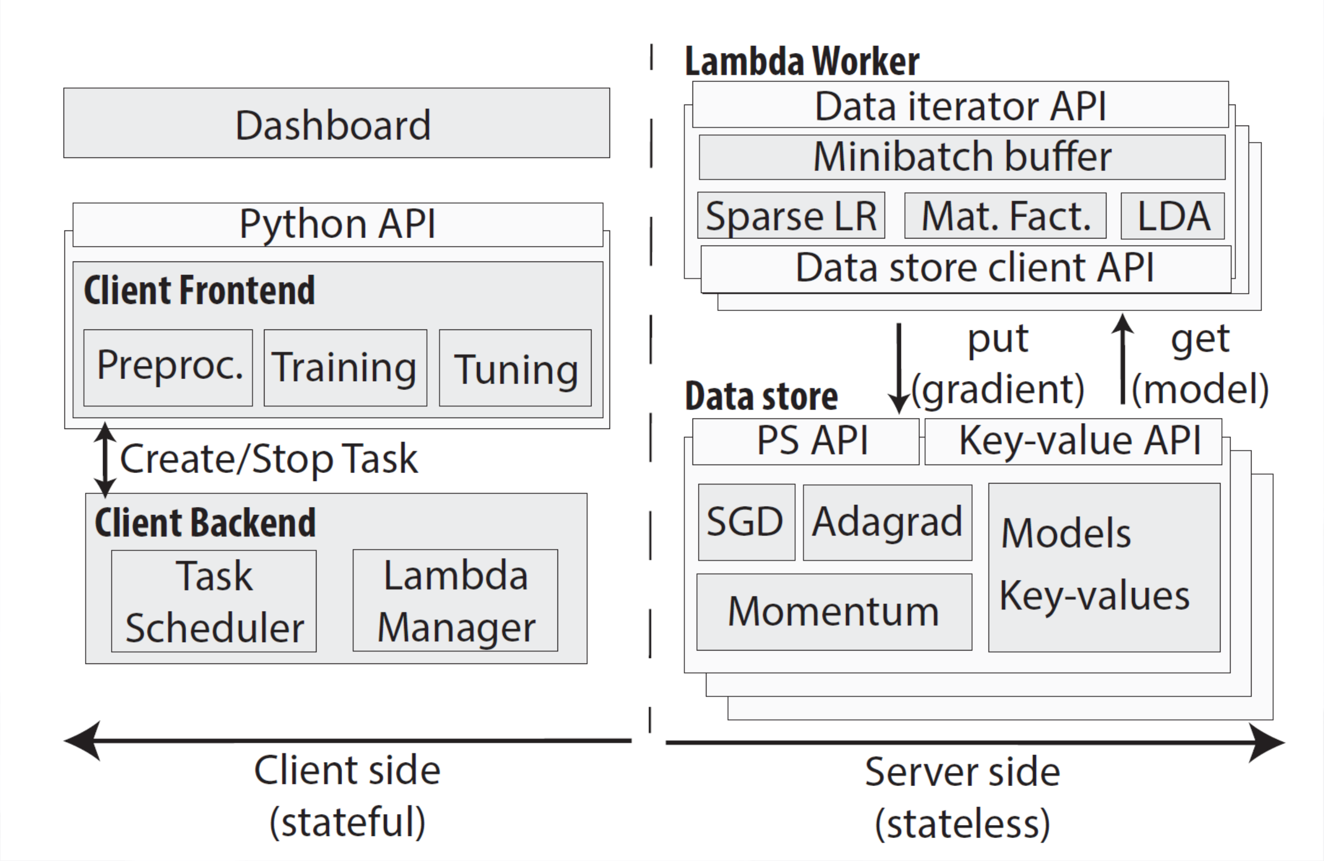
Design: Framework

Client Side

- Client Frontend
- Client Backend

Server Side

- Lambda Worker
- Data Store



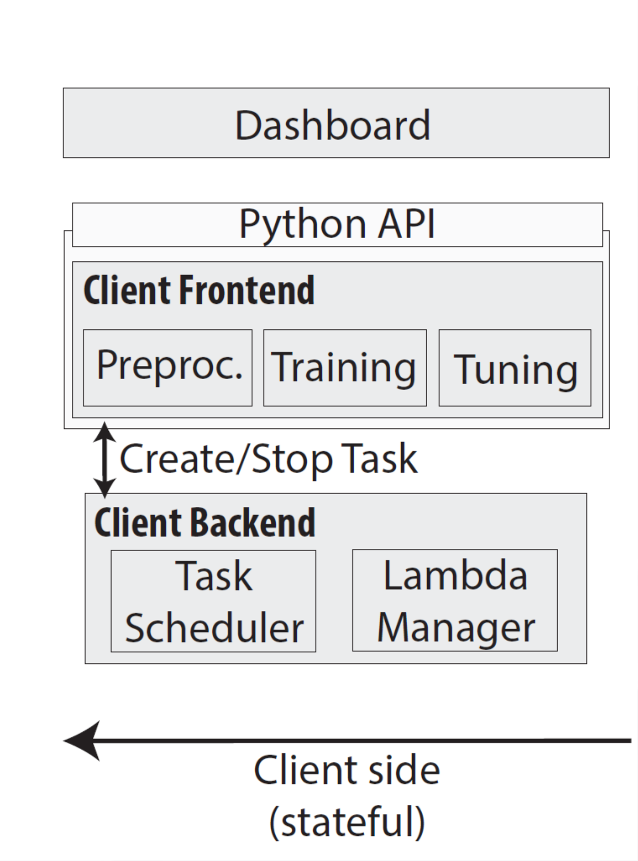
Design: Client Side

Python frontend

- Preprocessing
- Training
- Hyperparameter optimization

Client-side backend

- parse training data and load it to S3
- launch the Cirrus workers on lambdas
- manage the distributed data store
- keep track of the progress of computations
- return results to the Python frontend



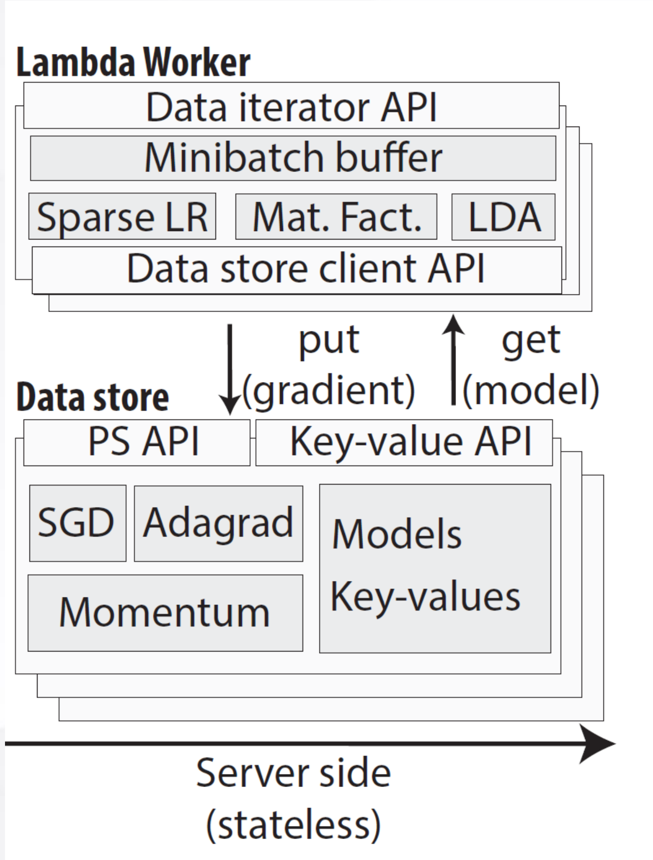
Design: Server Side

Worker runtime

- a smart iterator for training datasets stored in S3
- provides an API for the distributed data store

Distributed data store

API	Description
int send_gradient_X(ModelGradient* g)	Sends model gradient
SparseModel get_sparse_model_X(const std::vector<int>& indices)	Get subset of model
Model get_full_model_X()	Get all model weights
set_value(string key, char* data, int size)	Set intermediate state
std::string get_value(string key)	Get intermediate state



Design: End-to-end Workflow

```python
import cirrus
import numpy as np

local_path = "local_criteo"
s3_input = "criteo_dataset"
s3_output = "criteo_norm"

cirrus.load_libsvm(local_path, s3_input)

cirrus.normalize(s3_input, s3_output,
                 MIN_MAX_SCALING)
```

(a) Pre-process

```python
params = {
 'n_workers': 5,
 'n_ps': 1,
 'worker_size': 1024,
 'dataset': s3_output,
 'epsilon': 0.0001,
 'timeout': 20 * 60,
 'model_size': 2**19,
}

lr_task =  cirrus.LogisticRegression(params)
result = lr_task.run()
```

(b) Train

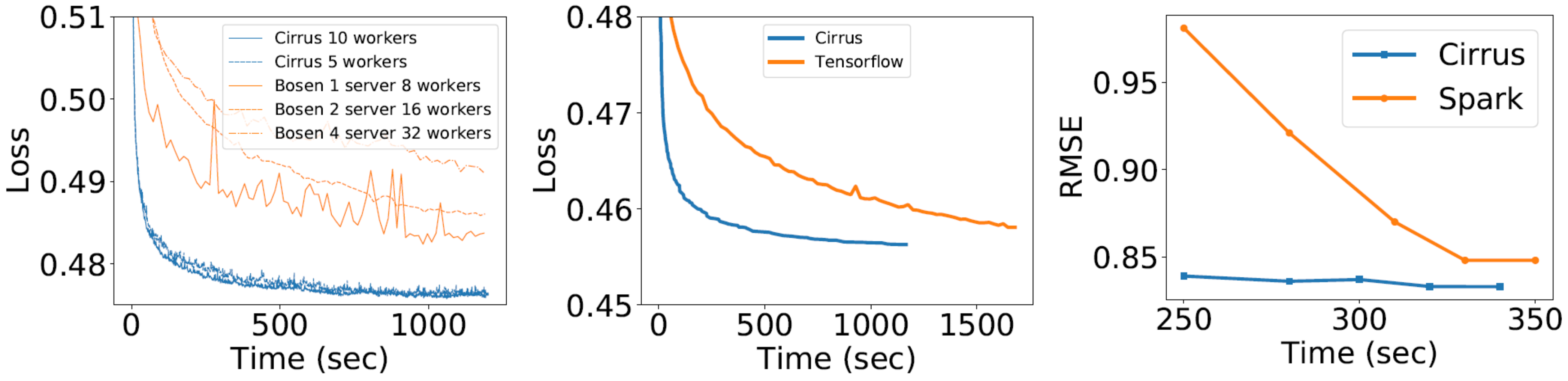
```python
# learning rates
lrates = np.arange(0.1, 10, 0.1)
minibatch_size = [100, 1000]

gs = cirrus.GridSearch(
 task=cirrus.LRegression,
 param_base=params,
 hyper_vars=["learning_rate", "minibach_size"],
 hyper_params=[lrates, minibatch_size])

results = gs.run()
```

(c) Tune

Evaluation: Sparse Logistic Regression

Baseline

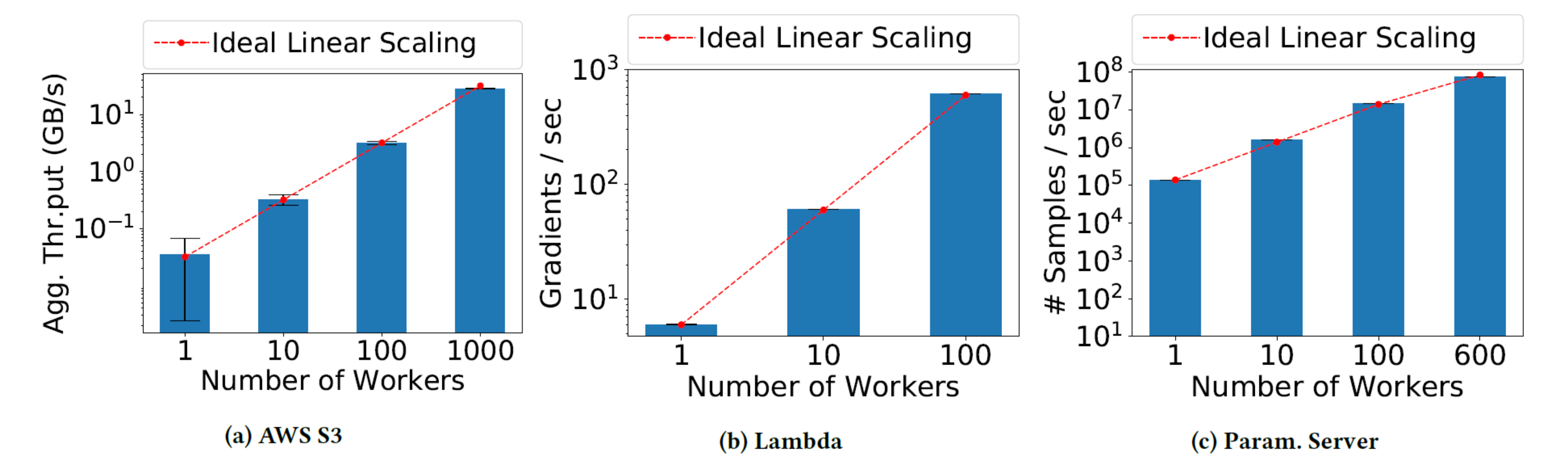
- Bosen
- TensorFlow
- Spark



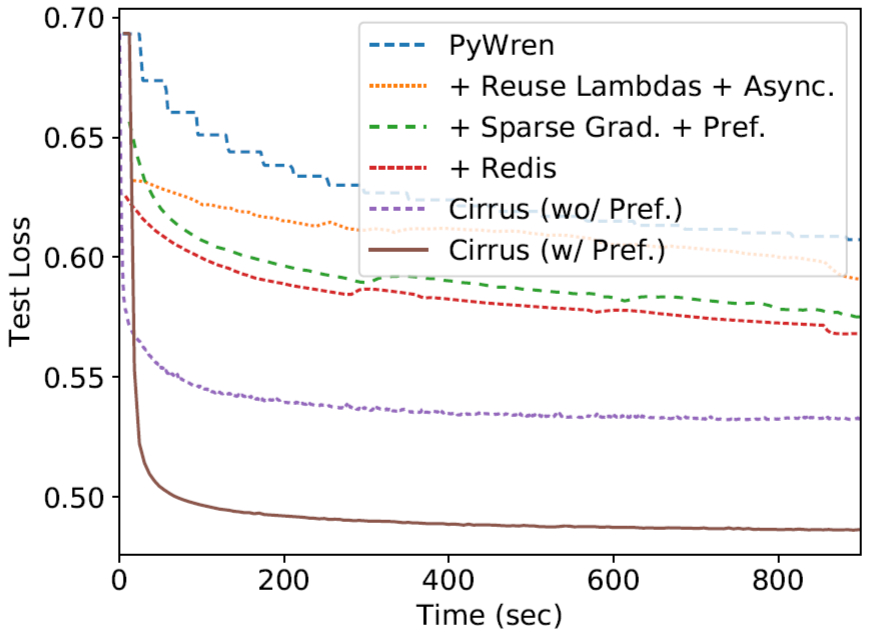
(a) Bosen (b) Tensorflow (c) Spark

Evaluation: Scalability

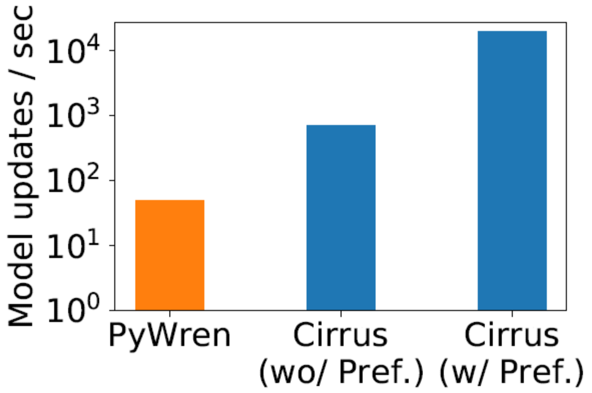
- Storage scalability
- Compute scalability
- Parameter server scalability



Evaluation: The Benefits of ML Specialization



(a) Convergence over time.



(b) Model updates per second.

03

LambdaML

- Towards Demystifying Serverless Machine Learning Training

Background: Distributed Machine Learning

Data and Model

Optimization Algorithm

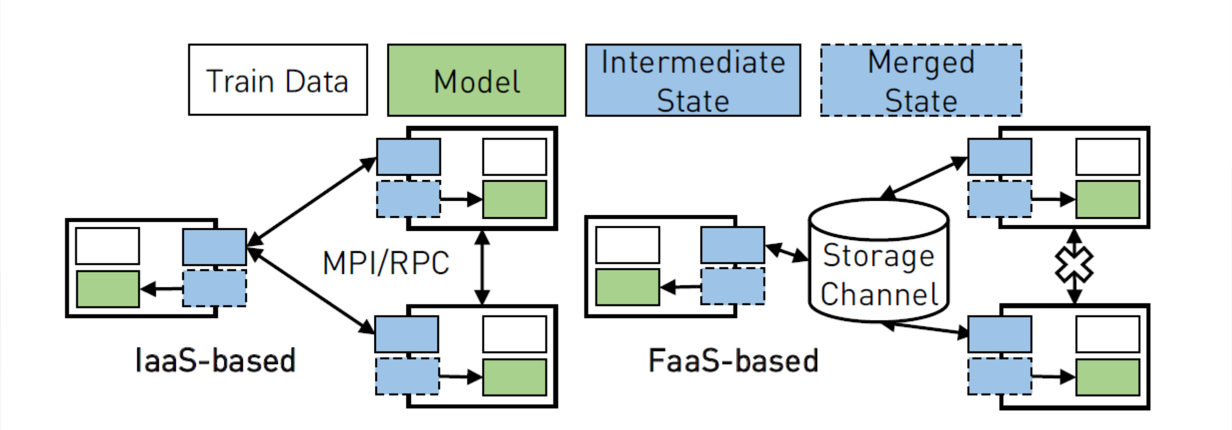
- In each iteration, the training procedure would typically scan the training data, compute necessary quantities (e.g., gradients), and update the model.
- Training ML models in a distributed setting is more complex, due to the extra complexity of distributed computation as well as coordination of the communication between executors.

Communication Mechanism

- **Communication Channel:** The efficiency of data transmission relies on the underlying communication channel.
- **Communication Pattern:** Gather, AllReduce, and ScatterReduce
- **Synchronization Protocol:** bulk synchronous parallel (BSP), asynchronous parallel (ASP)

Background: FaaS vs. IaaS for ML

- **IaaS: users have to build a cluster by renting VMs or reserve a cluster with predetermined configuration parameters**
  - Cons: There is no elasticity or auto-scaling if the reserved computation resources turn out to be insufficient.
- **FaaS**
  - Pros: Resource allocation in FaaS is on-demand and auto-scaled, and users are only charged by their actual resource usages.
  - Cons: FaaS currently does not support customized scaling and scheduling strategies.



Design: System Overview

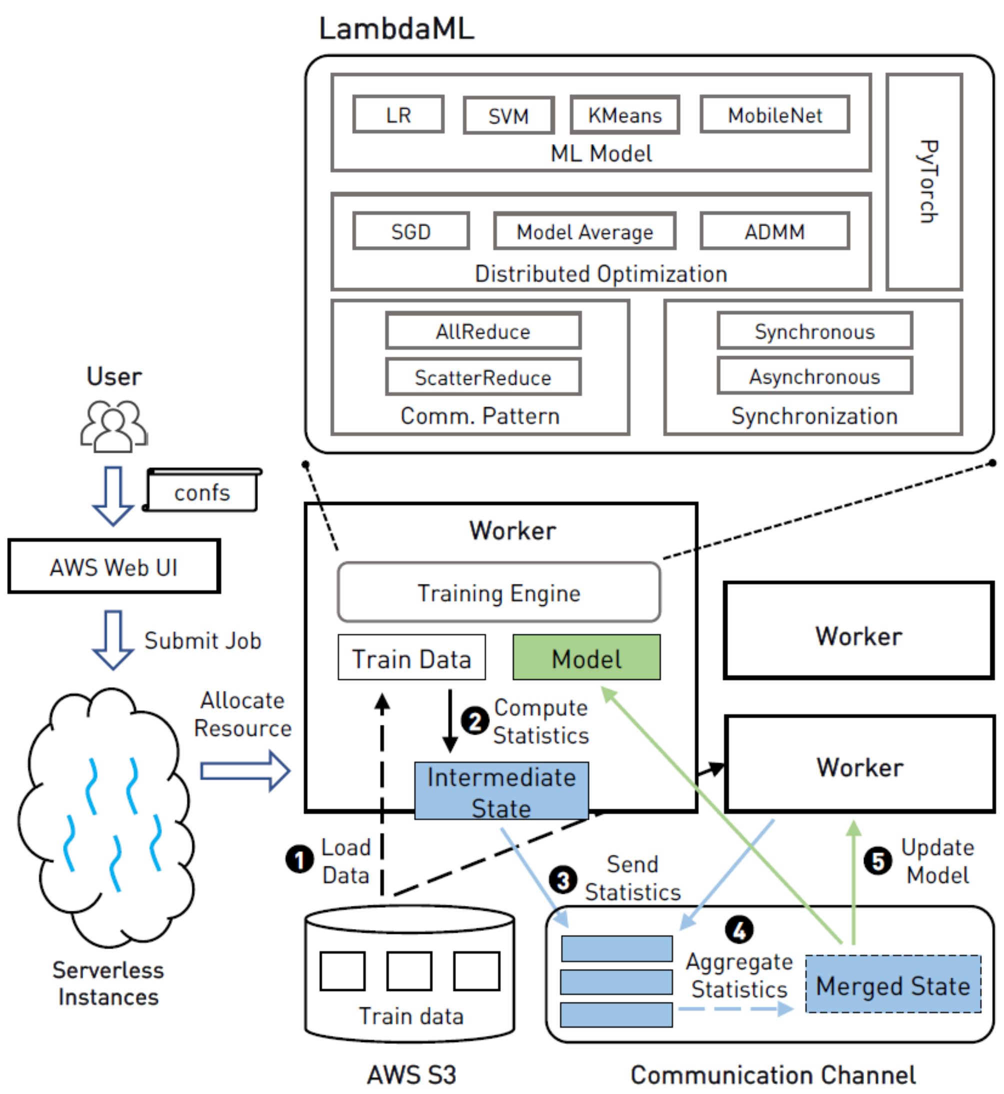
① **Load data**

② **Compute statistics**

③ **Send statistics**

④ **Aggregate statistics**

⑤ **Update model**



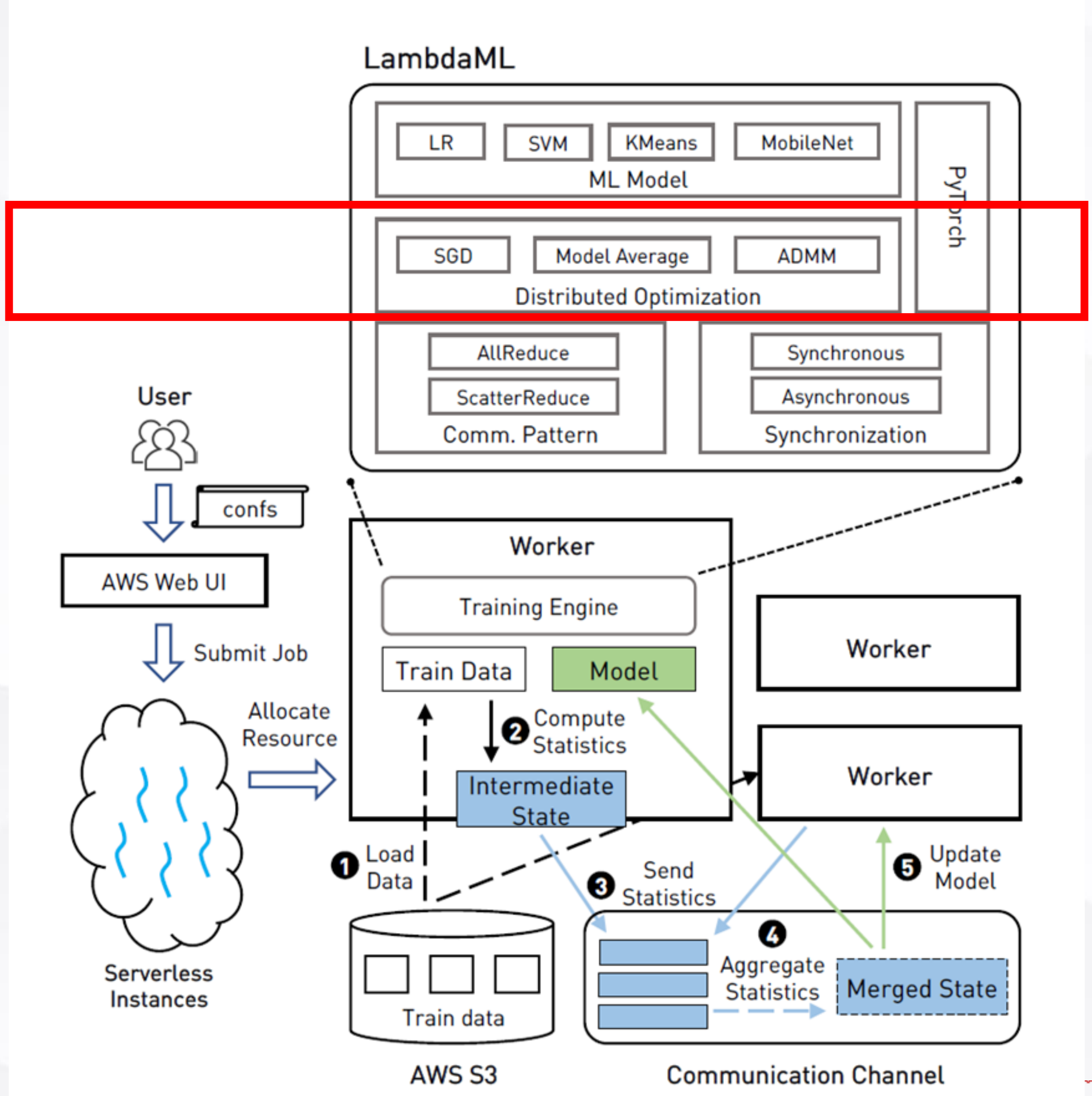
Design: Distributed Optimization Algorithm

Distributed SGD

- Stochastic gradient descent (SGD) is perhaps the most popular optimization algorithm.
- **Gradient Averaging:** GA updates the global model in every iteration by harvesting and aggregating the (updated) gradients from the executors.
- **Model Averaging:** MA collects and aggregates the (updated) local models.

Distributed ADMM

- ADMM breaks a large-scale convex optimization problem into several smaller subproblems



Design: Communication Channel

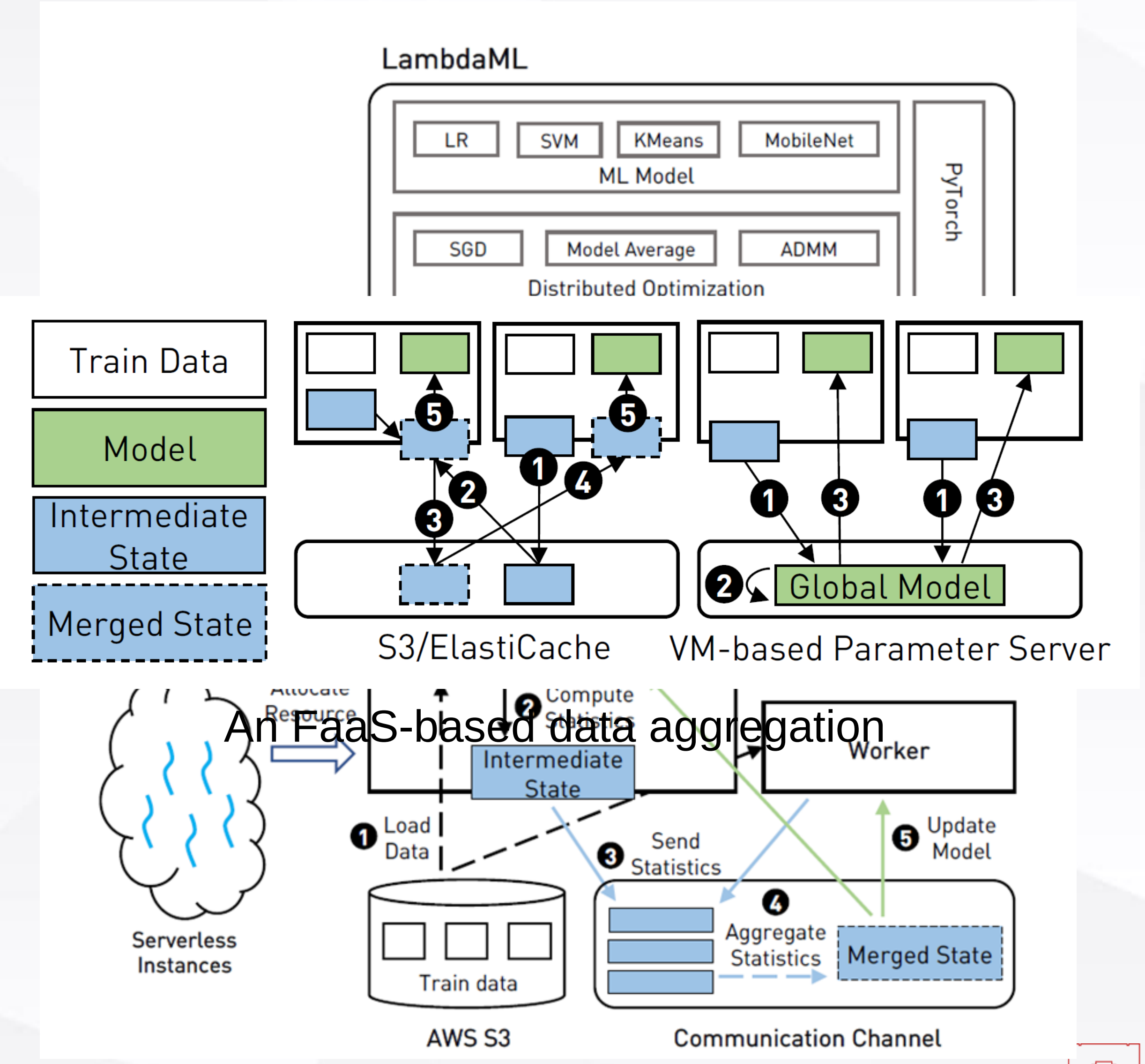
① Each executor stores its generated intermediate data as a temporary file in S3;

② The first executor pulls all temporary files from the storage service and merges them to a single file;

③ The leader writes the merged file back to the storage service;

④ All the other executors (except the leader) read the merged file from the storage service;

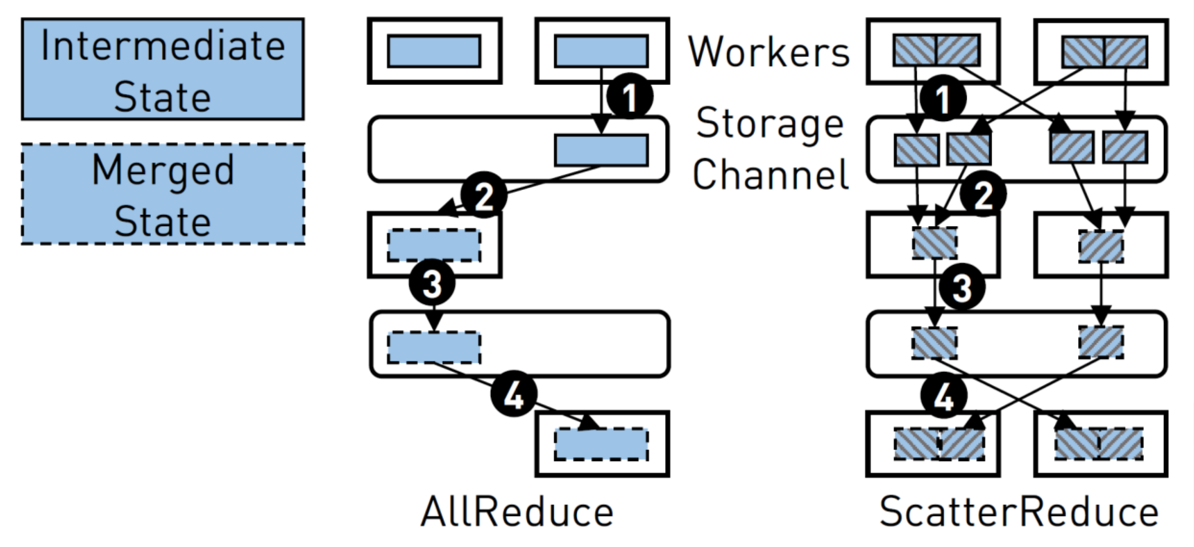
⑤ All executors refresh their (local) model with information read from the merged file.

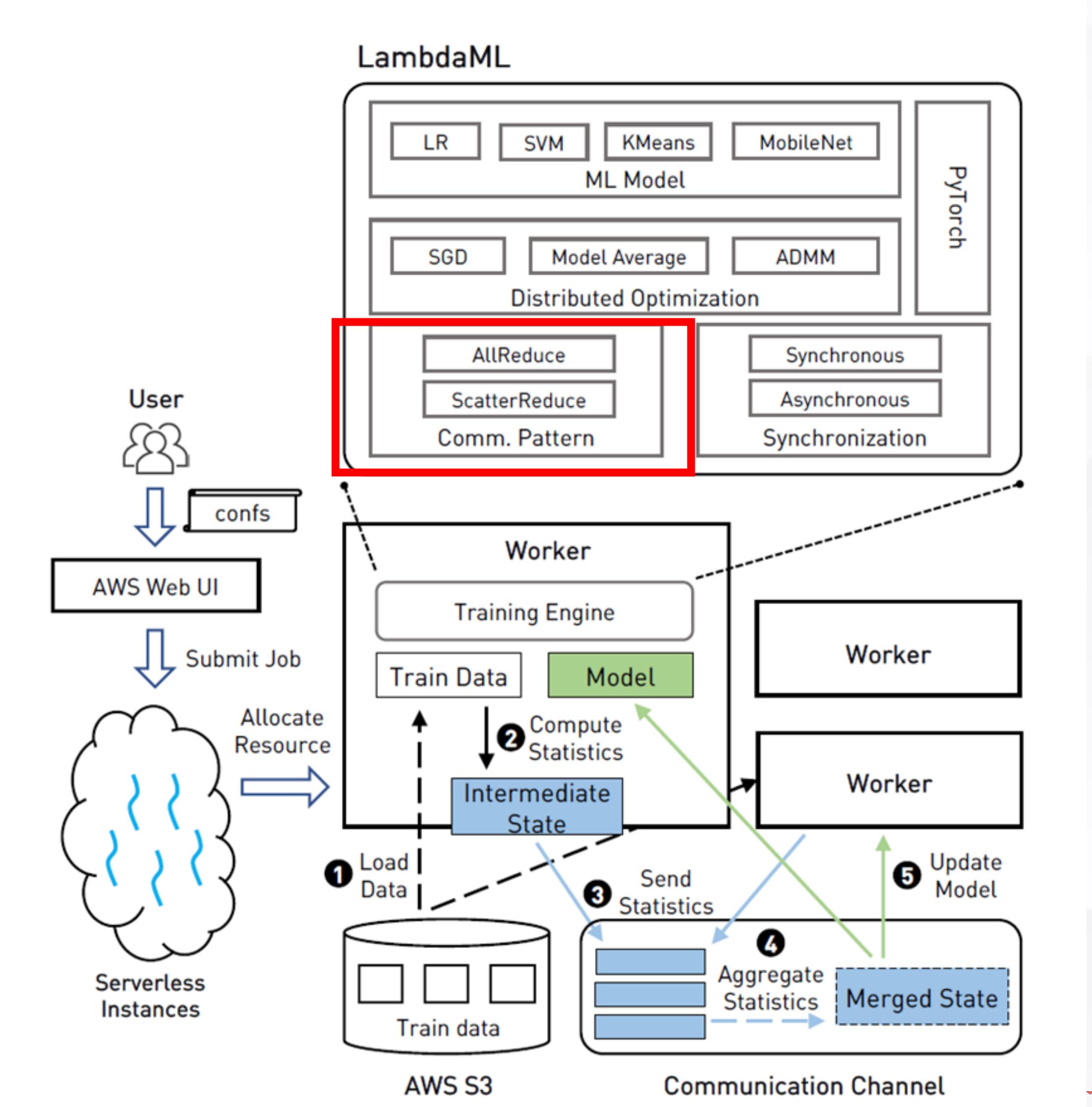


An FaaS-based data aggregation

Design: Communication Pattern

- AllReduce
- ScatterReduce





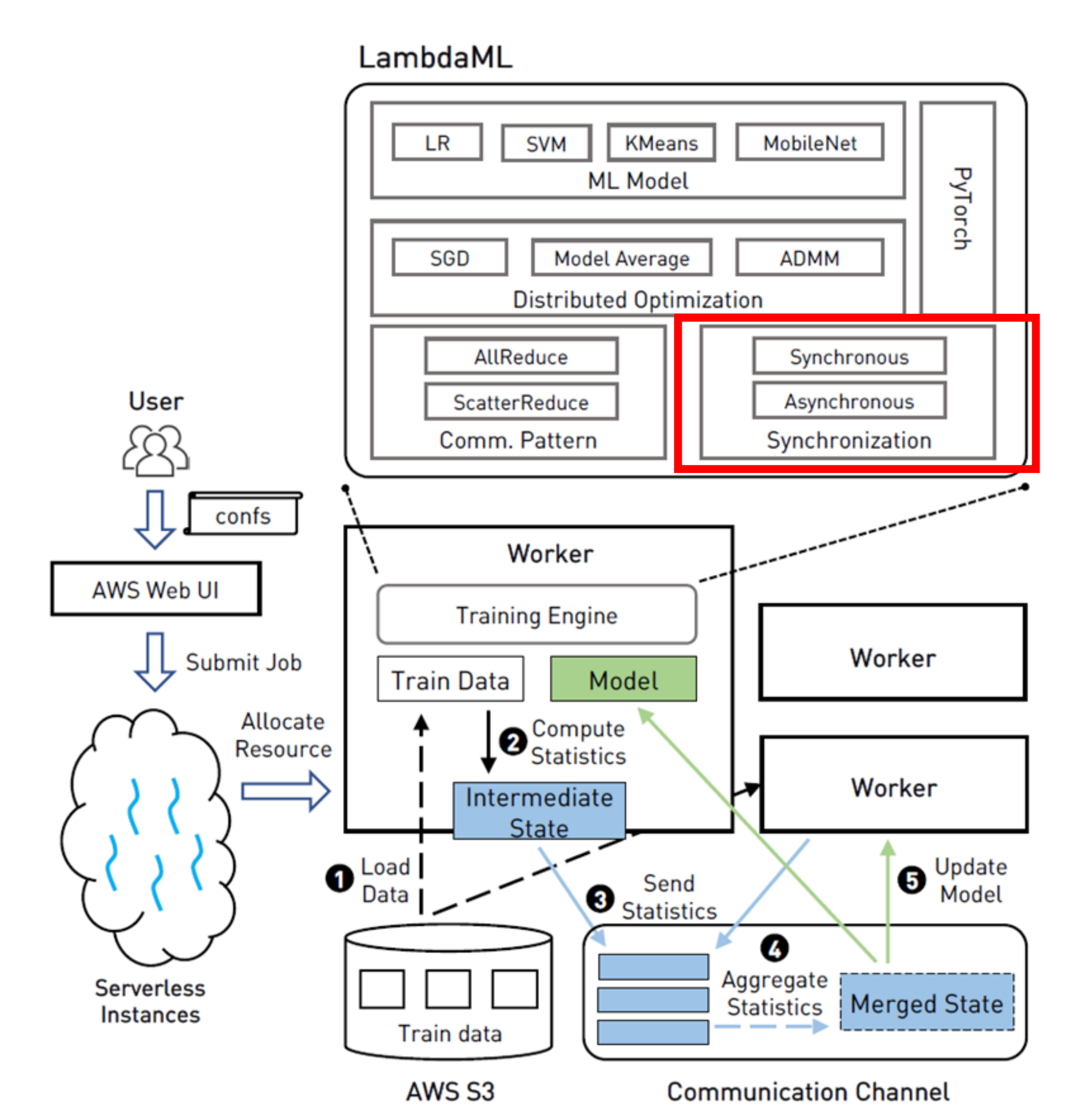
Design: Synchronization Protocol

Synchronous

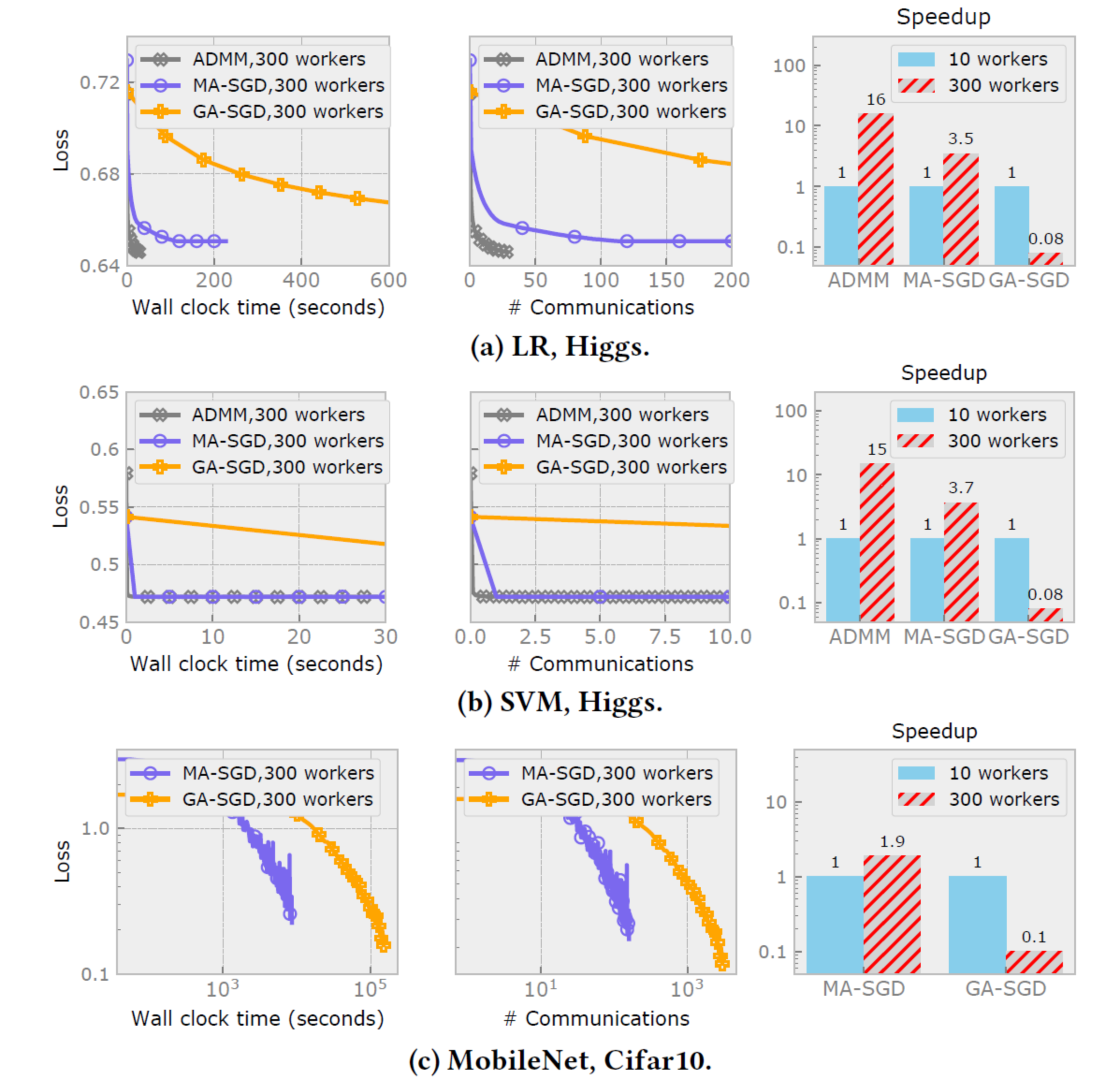
- Merging phase: All executors first write their local updates to the storage service. The reducer/aggregator waits all the other executors.
- Updating phase: The aggregator finishes aggregating all data and stores the aggregated information back to the storage service.

Asynchronous

- One replica of the trained model is stored on the storage service as a global state.
- Each executor runs independently – it reads the model from the storage service, updates the model with training data, writes
- the new model back to the storage service – without caring about the speeds of the other executors.



Evaluation: Distributed Optimization Algorithm



(a) LR, Higgs.

(b) SVM, Higgs.

(c) MobileNet, Cifar10.

Evaluation: Communication Channel

- Comparison of S3, Memcached, DynamoDB, and VM-based parameter server.
- A relative cost larger than 1 means S3 is cheaper, whereas a slowdown larger than 1 means S3 is faster.
- DynamoDB cannot handle a large model such as MobileNet.

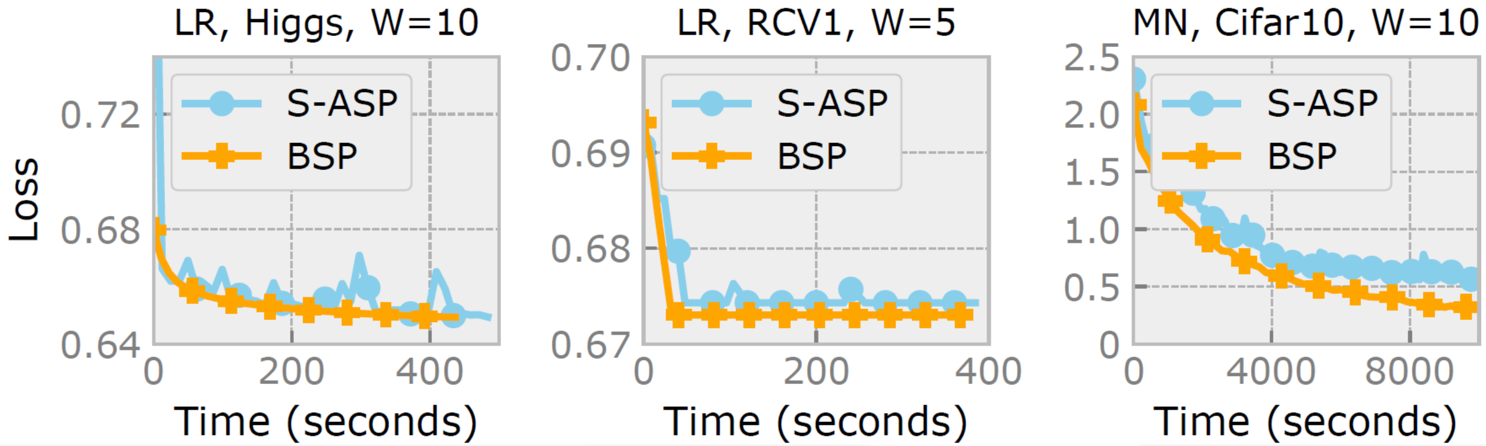
Workload	Memcached vs. S3		DynamoDB vs. S3		VM-PS vs. S3	
	cost	slowdown	cost	slowdown	cost	slowdown
LR,Higgs,W=10	5	4.17	0.95	0.83	4.7	3.85
LR,Higgs,W=50	4.5	3.70	0.92	0.81	4.47	3.70
KMeans,Higgs,W=50,k=10	1.58	1.32	1.13	0.93	1.48	1.23
KMeans,Higgs,W=50,k=1K	1.43	1.19	1.03	0.90	1.52	1.27
MobileNet,Cifar10,W=10	0.9	0.77	N/A	N/A	4.8	4.01
MobileNet,Cifar10,W=50	0.89	0.75	N/A	N/A	4.85	4.03

Evaluation

Communication Patterns

Model & Dataset	Model Size	AllReduce	ScatterReduce
LR,Higgs,W=50	224B	9.2s	9.8s
MobileNet,Cifar10,W=10	12MB	3.3s	3.1s
ResNet,Cifar10,W=10	89MB	17.3s	8.5s

Synchronization Protocols



04

INFless

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

INFless’ s Overview

Background: Existing serverless platforms do not cater to the needs of ML inference.

- do not address the challenge of providing solutions for guaranteeing latency
- the resource efficiency at the serverless provider side is also very low

Design Goal: A native serverless inference system introduces several challenges that need to be addressed.

- Low latency
- High throughput
- Low overhead

Contribution

- 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.
- We design a novel **Long-Short Term Histogram (LSTH)** policy.
- We completely implement INFless based on OpenFaaS.

Background: Limitations of Existing Serverless Platforms

- **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.
- **Observation #2: For batch-enabled inference, commercial serverless platforms cannot provide low-latency services for some small-sized models.**
- **Observation #3: Resource over-provisioning**
  - The proportional CPU-memory allocation policy set by a commercial serverless platform does not fit with computationally-intensive inference.
- **Observation #4: The "one-to-one mapping" request processing policy of commercial serverless platforms causes low resource utilization.**
- **Observation #5: OTP batching lacks the codesign of batch configuration, instance scheduling and resource allocation, bringing only limited throughput improvement.**

Design: System Architecture

① Function deployment

② DAG structure parsing

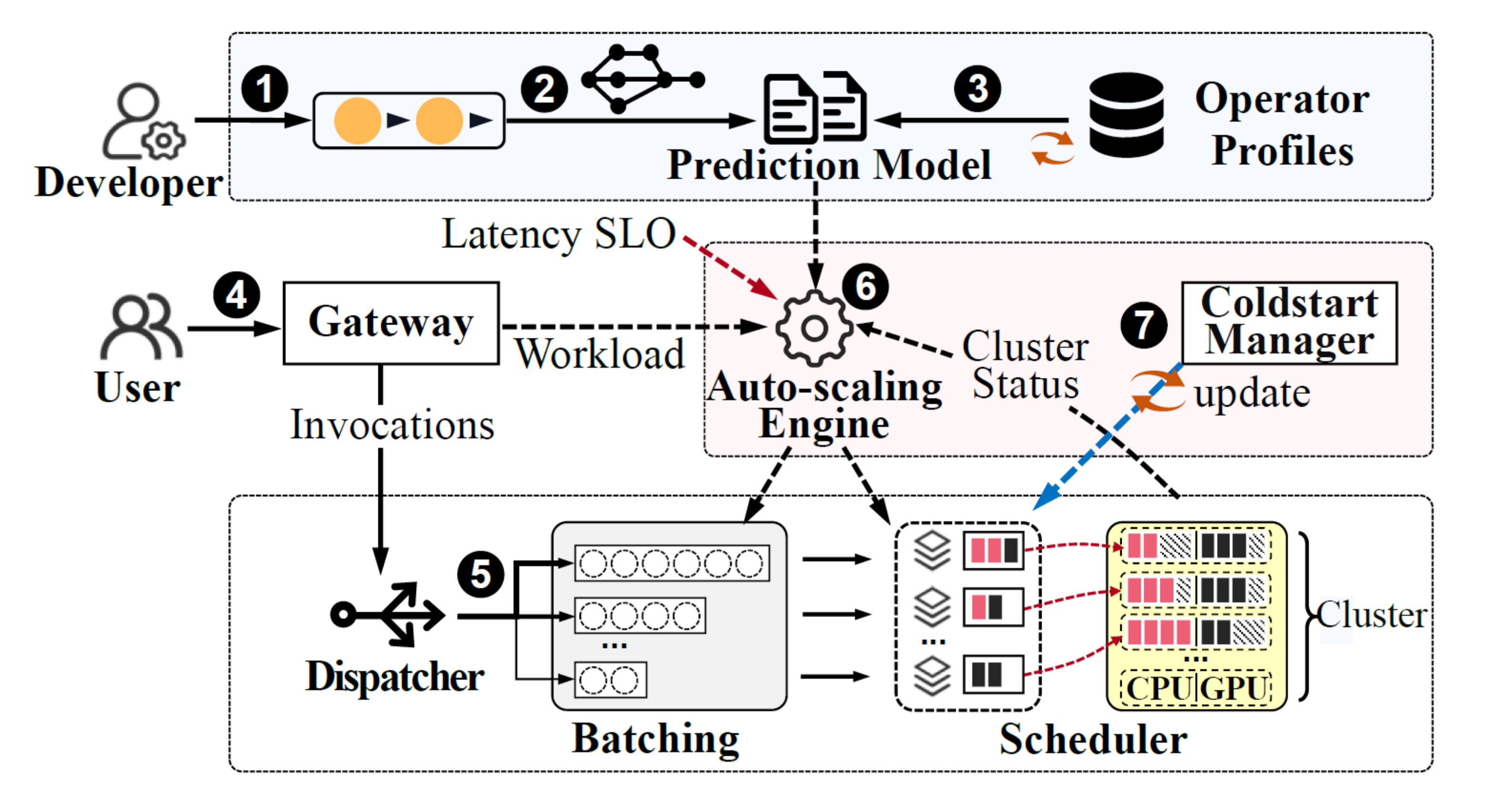
③ **Operator profiling**

④ Inference query

⑤ **Dispatching and batching**

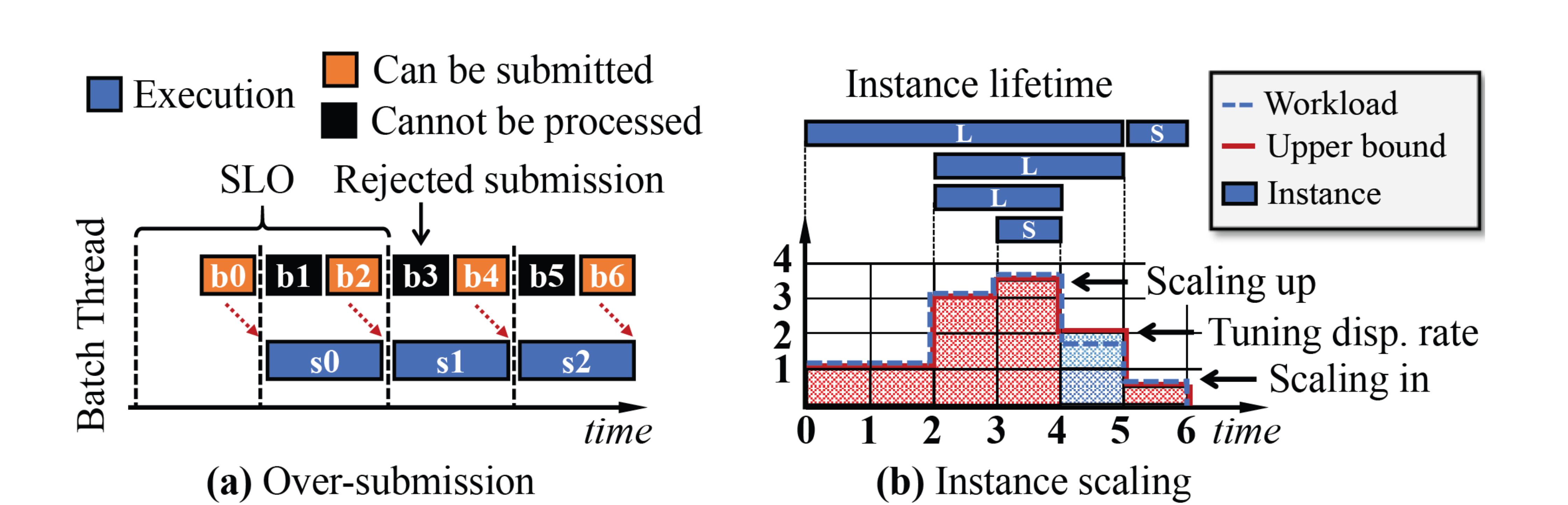
⑥ Resource configuration

⑦ **Cold-start avoidance**



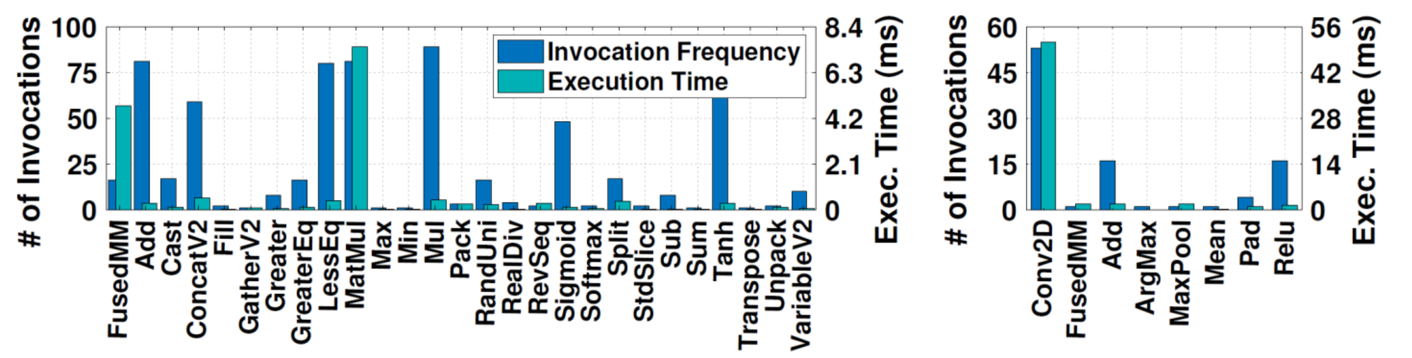
Design: Built-in, Non-Uniform Batching

- **Built-in:** Batching is integrated into the serverless platform, enabling simultaneous, collaborative control over batch size, resource allocation and placements.
- **Non-uniform:** Each instance has an individual batch queue to aggregate inference requests.



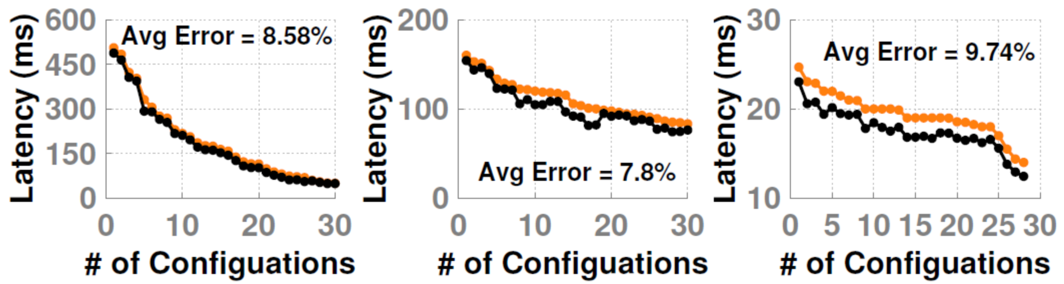
Design: Combined Operator Profiling

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



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

- Database: build a operator profile database <operator, batch-size, CPU, GPU, time>, and estimate the model execution latency based on the database.
- Result

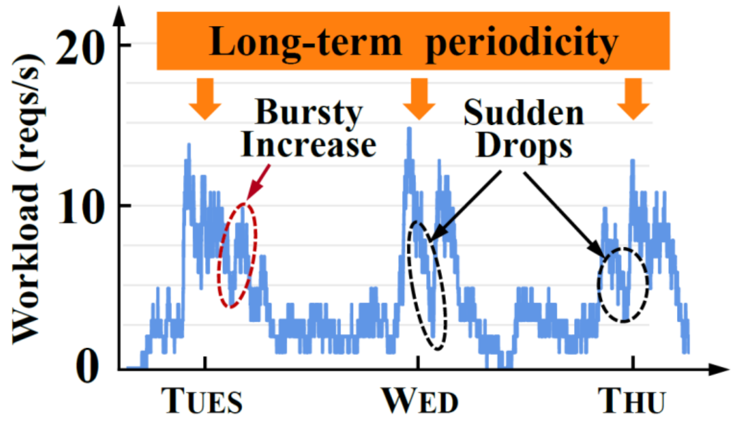


(a) ResNet-50 (b) MobileNet (c) LSTM-2365

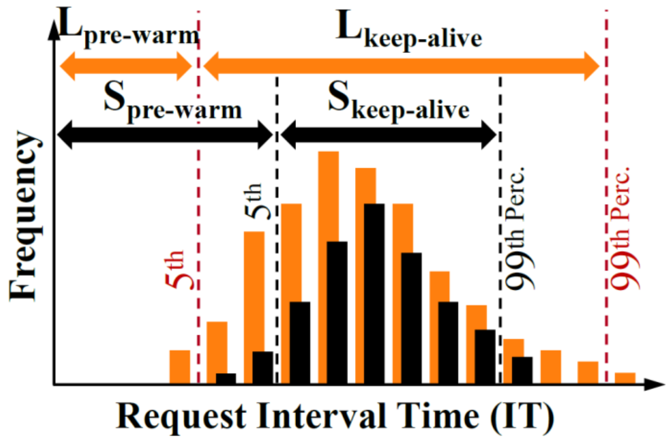
Design: Managing Cold Starts with LSTH

- **Long-term periodicity (LTP): the request load shows a diurnal user access pattern overall;**
- **Short-term burst (STB): there are many sudden changes (including both increases and decreases) in short times.**
- **Long-Short Term Histogram (LSTH)**

$$\text{pre-warm} = \gamma L_{\text{pre-warm}} + (1-\gamma) S_{\text{pre-warm}}$$
$$\text{keep-alive} = \gamma L_{\text{keep-alive}} + (1-\gamma) S_{\text{keep-alive}}$$



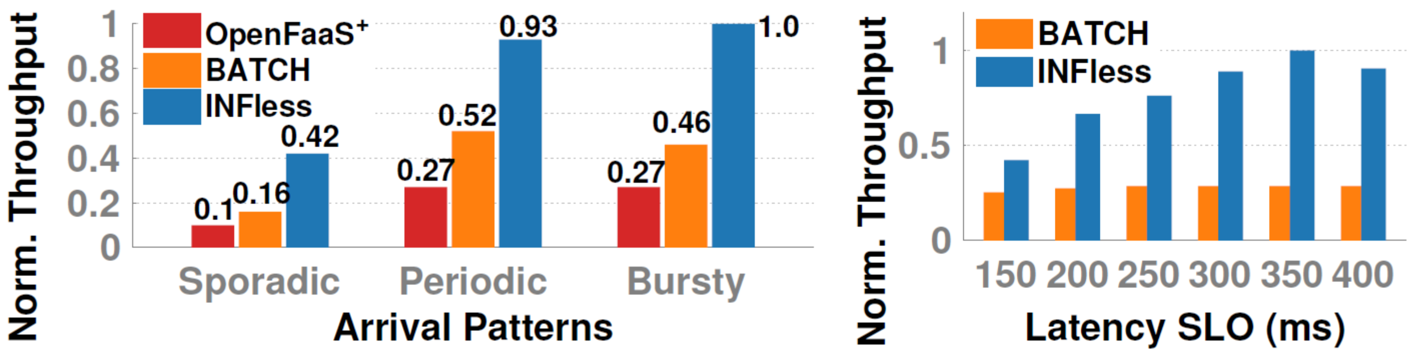
(a) Workload features



(b) Weighted hybrid hist policy

Evaluation: Local Cluster Evaluation

- High throughput: INFless improves system throughput by 2x-5x.



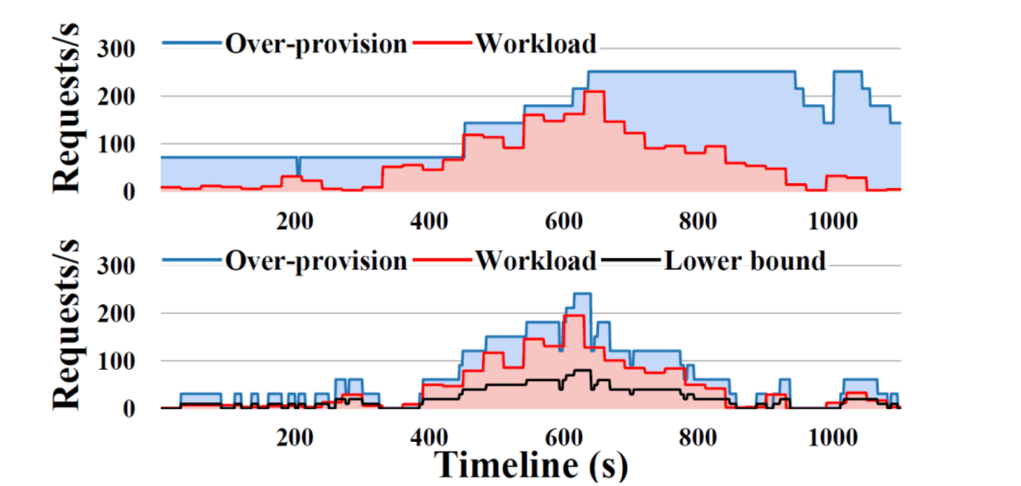
(a) Thp. under production traces (b) Thp. under different SLOs

- Component analysis: Every component of INFless contributes much to throughput improvement, with batching being the highest.
- Flexible configurations: INFless opts for flexible configurations on both batch-sizes and resource allocations.

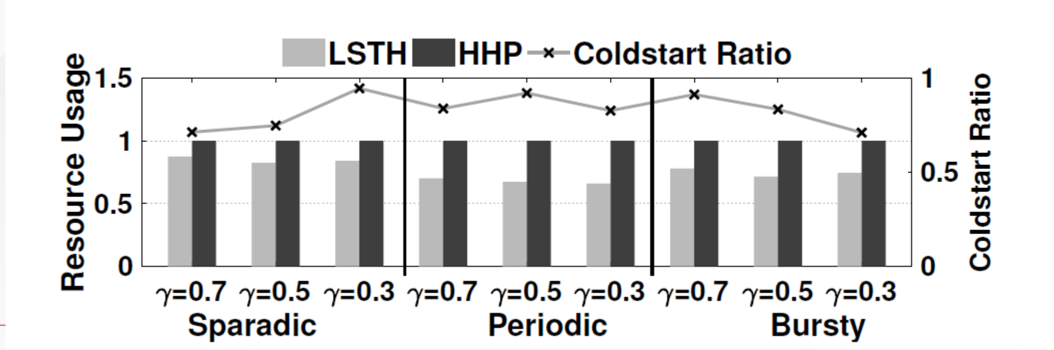
SJTU

Evaluation: Local Cluster Evaluation

- Less over-provisioning: INFless's resource allocation policy reduces the resource provisioning significantly.

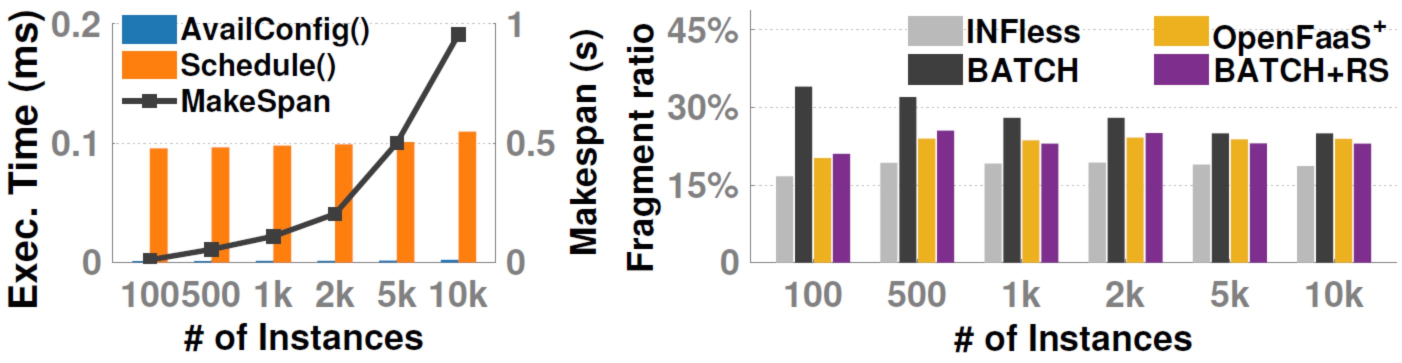


- SLO violation: INFless can guarantee the latency SLO of inference workloads.
- Cold start: Compared with HHP, our LSTH policy can reduce the cold start rate by 20%.



Evaluation: Large Scale Simulation

- **Scalability: INFless scales well in large-scale evaluations.**
- **Resource fragments: INFless's resource-aware scheduling algorithm reduces the resource fragments significantly.**



(a) Scheduling overhead (b) Resource fragments

- **Cost efficiency: INFless can help service developers and cloud providers reduce the cost of constructing inference services.**

05

Conclusion

Serverless and Machine Learning

Paper	Year	Conference	Topic
NumPyWren	2020	SoCC	Matrix computation
Cirrus	2019	SoCC	Model training
LambdaML	2021	SIGMOD	Model training
INFless	2022	ASPLOS	Model inference

感谢聆听

饮 水 思 源　爱 国 荣 校