Scalable Input Data Processing for Resource-Efficient ML

Ana Klimovic

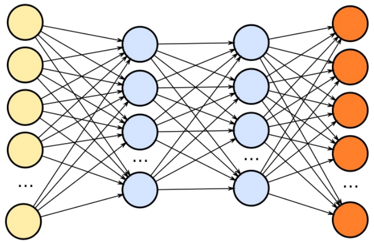
ETH zürich

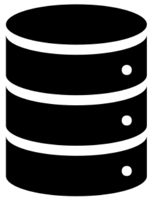
SoCC Keynote
November 2022

It's an exciting time for ML systems

Large growth in…

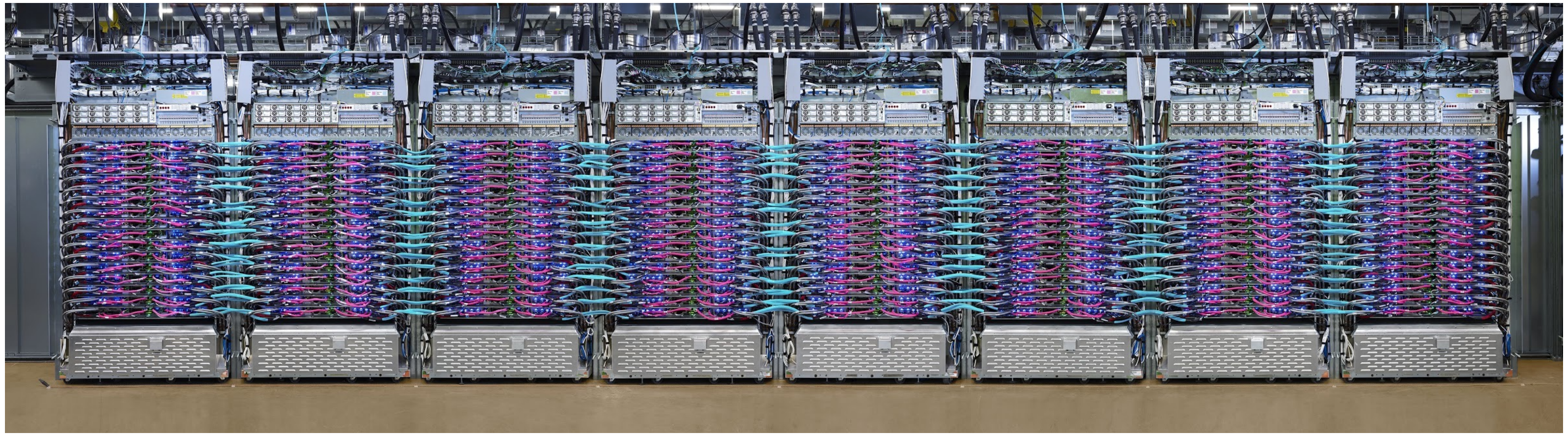
- ML use-cases
- ML model sizes
- Training data volume





Large growth in…

- ML use-cases
- ML model sizes
- Training data volume
- FLOPS provided by specialized ML hardware accelerators



Example: Google TPU pod

Large growth in…

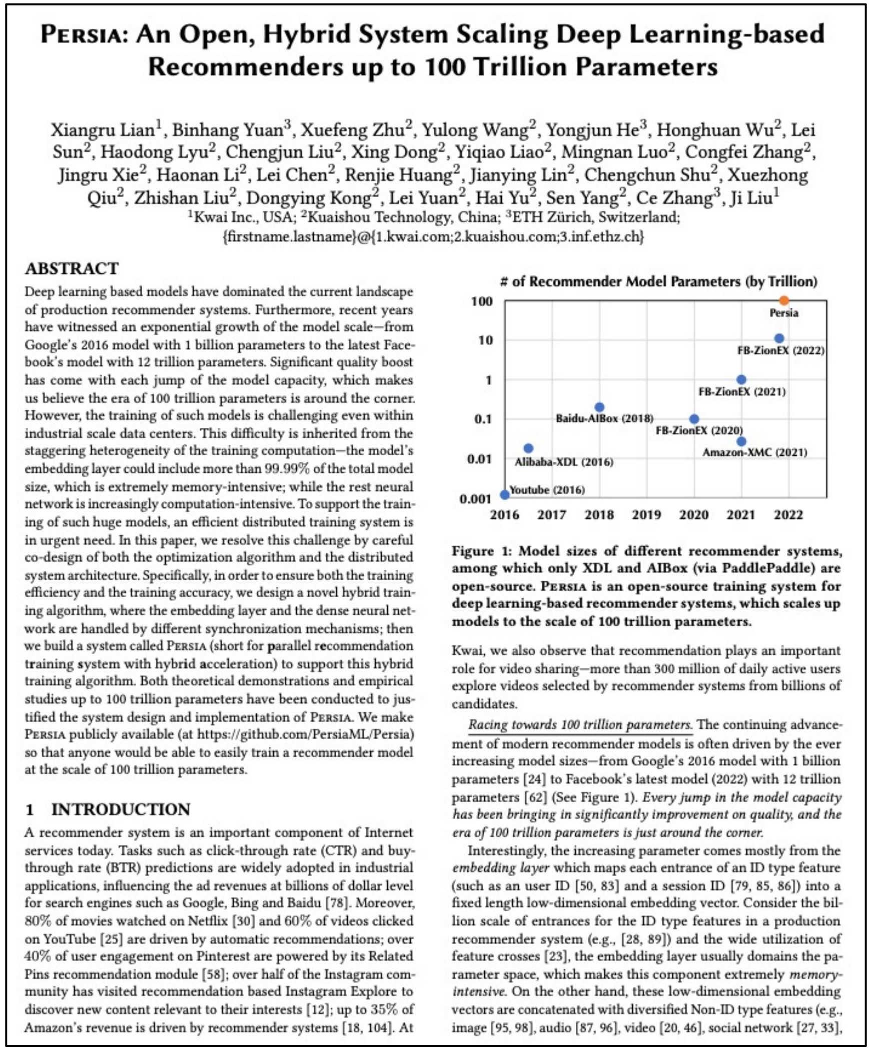
- ML use-cases
- ML model sizes
- Training data volume
- FLOPS provided by specialized ML hardware accelerators
- **Cost!!!**

→ Training ML models consumes many GPU/TPU-hours and $$$

How much does it cost…

…to train a 100 Trillion parameter model for 1 day on the cloud?

A. $4,000
B. $40,000
C. $400,000



Persia: An Open, Hybrid System Scaling Deep Learning-based Recommenders up to 100 Trillion Parameters

Xiangru Lian[1], Binhang Yuan[3], Xuefeng Zhu[2], Yulong Wang[2], Yongjun He[3], Honghuan Wu[2], Lei Sun[2], Haodong Lyu[2], Chengjun Liu[2], Xing Dong[2], Yiqiao Liao[2], Mingnan Luo[2], Congfei Zhang[2], Jingru Xie[2], Haonan Li[2], Lei Chen[2], Renjie Huang[2], Jianying Lin[2], Chengchun Shu[2], Xuezhong Qiu[2], Zhishan Liu[2], Dongying Kong[2], Lei Yuan[2], Hai Yu[2], Sen Yang[2], Ce Zhang[3], Ji Liu[1]
[1]Kwai Inc., USA; [2]Kuaishou Technology, China; [3]ETH Zürich, Switzerland;
{firstname.lastname}@{1.kwai.com;2.kuaishou.com;3.inf.ethz.ch}

ABSTRACT

Deep learning based models have dominated the current landscape of production recommender systems. Furthermore, recent years have witnessed an exponential growth of the model scale—from Google's 2016 model with 1 billion parameters to the latest Facebook's model with 12 trillion parameters. Significant quality boost has come with each jump of the model capacity, which makes us believe the era of 100 trillion parameters is around the corner. However, the training of such models is challenging even within industrial scale data centers. This difficulty is inherited from the staggering heterogeneity of the training computation—the model's embedding layer could include more than 99.99% of the total model size, which is extremely memory-intensive; while the rest neural network is increasingly computation-intensive. To support the training of such huge models, an efficient distributed training system is in urgent need. In this paper, we resolve this challenge by careful co-design of both the optimization algorithm and the distributed system architecture. Specifically, in order to ensure both the training efficiency and the training accuracy, we design a novel hybrid training algorithm, where the embedding layer and the dense neural network are handled by different synchronization mechanisms; then we build a system called Persia (short for parallel recommendation training system with hybrid acceleration) to support this hybrid training algorithm. Both theoretical demonstrations and empirical studies up to 100 trillion parameters have been conducted to justified the system design and implementation of Persia. We make Persia publicly available (at https://github.com/PersiaML/Persia) so that anyone would be able to easily train a recommender model at the scale of 100 trillion parameters.

1 INTRODUCTION

A recommender system is an important component of Internet services today. Tasks such as click-through rate (CTR) and buy-through rate (BTR) predictions are widely adopted in industrial applications, influencing the ad revenues at billions of dollar level for search engines such as Google, Bing and Baidu [78]. Moreover, 80% of movies watched on Netflix [30] and 60% of videos clicked on YouTube [25] are driven by automatic recommendations; over 40% of user engagement on Pinterest are powered by its Related Pins recommendation module [58]; over half of the Instagram community has visited recommendation based Instagram Explore to discover new content relevant to their interests [12]; up to 35% of Amazon's revenue is driven by recommender systems [18, 104]. At

# of Recommender Model Parameters (by Trillion)
100, 10, 1, 0.1, 0.01, 0.001
2016 2017 2018 2019 2020 2021 2022
Persia
FB-ZionEX (2022)
FB-ZionEX (2021)
Baidu-AIBox (2018)
FB-ZionEX (2020)
Amazon-XMC (2021)
Alibaba-XDL (2016)
Youtube (2016)

Figure 1: Model sizes of different recommender systems, among which only XDL and AIBox (via PaddlePaddle) are open-source. Persia is an open-source training system for deep learning-based recommender systems, which scales up models to the scale of 100 trillion parameters.

Kwai, we also observe that recommendation plays an important role for video sharing—more than 300 million of daily active users explore videos selected by recommender systems from billions of candidates.

Racing towards 100 trillion parameters. The continuing advancement of modern recommender models is often driven by the ever increasing model sizes—from Google's 2016 model with 1 billion parameters [24] to Facebook's latest model (2022) with 12 trillion parameters [62] (See Figure 1). *Every jump in the model capacity has been bringing in significantly improvement on quality, and the era of 100 trillion parameters is just around the corner.*

Interestingly, the increasing parameter comes mostly from the *embedding layer* which maps each entrance of an ID type feature (such as an user ID [50, 83] and a session ID [79, 85, 86]) into a fixed length low-dimensional embedding vector. Consider the billion scale of entrances for the ID type features in a production recommender system (e.g., [28, 89]) and the wide utilization of feature crosses [23], the embedding layer usually domains the parameter space, which makes this component extremely *memory-intensive.* On the other hand, these low-dimensional embedding vectors are concatenated with diversified Non-ID type features (e.g., image [95, 98], audio [87, 96], video [20, 46], social network [27, 33],

Slide from Ce Zhang

How much does it cost…

…to train a 100 Trillion parameter model for 1 day on the cloud?

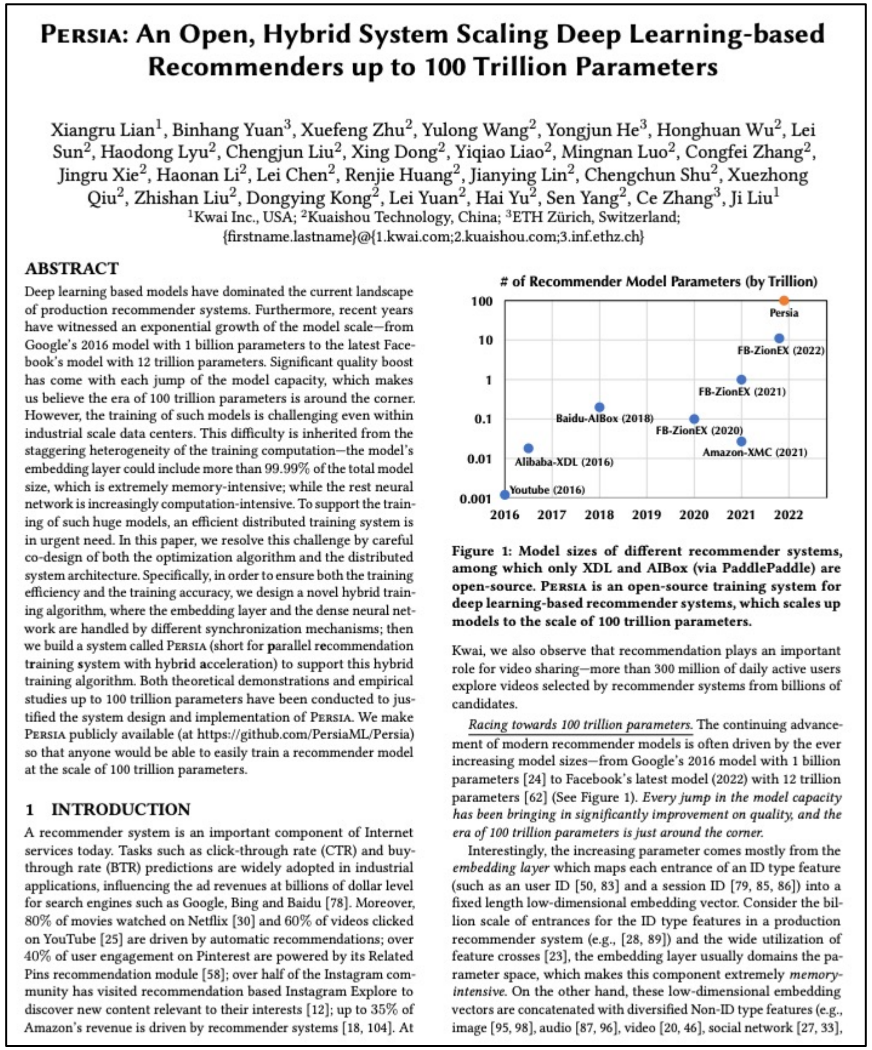
A. $4,000

B. $40,000

C. $400,000

Used:

- 3,000 CPU cores
- 64 A100 GPUs
- 360 TB of RAM



PERSIA: An Open, Hybrid System Scaling Deep Learning-based Recommenders up to 100 Trillion Parameters

Xiangru Lian[1], Binhang Yuan[3], Xuefeng Zhu[2], Yulong Wang[2], Yongjun He[3], Honghuan Wu[2], Lei Sun[2], Haodong Lyu[2], Chengjun Liu[2], Xing Dong[2], Yiqiao Liao[2], Mingnan Luo[2], Congfei Zhang[2], Jingru Xie[2], Haonan Li[2], Lei Chen[2], Renjie Huang[2], Jianying Lin[2], Chengchun Shu[2], Xuezhong Qiu[2], Zhishan Liu[2], Dongying Kong[2], Lei Yuan[2], Hai Yu[2], Sen Yang[2], Ce Zhang[3], Ji Liu[1]
[1]Kwai Inc., USA; [2]Kuaishou Technology, China; [3]ETH Zürich, Switzerland;
{firstname.lastname}@{1.kwai.com;2.kuaishou.com;3.inf.ethz.ch}

ABSTRACT

Deep learning based models have dominated the current landscape of production recommender systems. Furthermore, recent years have witnessed an exponential growth of the model scale—from Google's 2016 model with 1 billion parameters to the latest Facebook's model with 12 trillion parameters. Significant quality boost has come with each jump of the model capacity, which makes us believe the era of 100 trillion parameters is around the corner. However, the training of such models is challenging even within industrial scale data centers. This difficulty is inherited from the staggering heterogeneity of the training computation—the model's embedding layer could include more than 99.99% of the total model size, which is extremely memory-intensive; while the rest neural network is increasingly computation-intensive. To support the training of such huge models, an efficient distributed training system is in urgent need. In this paper, we resolve this challenge by careful co-design of both the optimization algorithm and the distributed system architecture. Specifically, in order to ensure both the training efficiency and the training accuracy, we design a novel hybrid training algorithm, where the embedding layer and the dense neural network are handled by different synchronization mechanisms; then we build a system called PERSIA (short for **p**arallel r**e**commendation t**r**aining **s**ystem with hybr**i**d **a**cceleration) to support this hybrid training algorithm. Both theoretical demonstrations and empirical studies up to 100 trillion parameters have been conducted to justified the system design and implementation of PERSIA. We make PERSIA publicly available (at https://github.com/PersiaML/Persia) so that anyone would be able to easily train a recommender model at the scale of 100 trillion parameters.

1 INTRODUCTION

A recommender system is an important component of Internet services today. Tasks such as click-through rate (CTR) and buy-through rate (BTR) predictions are widely adopted in industrial applications, influencing the ad revenues at billions of dollar level for search engines such as Google, Bing and Baidu [78]. Moreover, 80% of movies watched on Netflix [30] and 60% of videos clicked on YouTube [25] are driven by automatic recommendations; over 40% of user engagement on Pinterest are powered by its Related Pins recommendation module [58]; over half of the Instagram community has visited recommendation based Instagram Explore to discover new content relevant to their interests [12]; up to 35% of Amazon's revenue is driven by recommender systems [18, 104]. At

Figure 1: Model sizes of different recommender systems, among which only XDL and AIBox (via PaddlePaddle) are open-source. PERSIA is an open-source training system for deep learning-based recommender systems, which scales up models to the scale of 100 trillion parameters.

Kwai, we also observe that recommendation plays an important role for video sharing—more than 300 million of daily active users explore videos selected by recommender systems from billions of candidates.

Racing towards 100 trillion parameters. The continuing advancement of modern recommender models is often driven by the ever increasing model sizes—from Google's 2016 model with 1 billion parameters [24] to Facebook's latest model (2022) with 12 trillion parameters [62] (See Figure 1). *Every jump in the model capacity has been bringing in significantly improvement on quality, and the era of 100 trillion parameters is just around the corner.*

Interestingly, the increasing parameter comes mostly from the *embedding layer* which maps each entrance of an ID type feature (such as an user ID [50, 83] and a session ID [79, 85, 86]) into a fixed length low-dimensional embedding vector. Consider the billion scale of entrances for the ID type features in a production recommender system (e.g., [28, 89]) and the wide utilization of feature crosses [23], the embedding layer usually domains the parameter space, which makes this component extremely *memory-intensive*. On the other hand, these low-dimensional embedding vectors are concatenated with diversified Non-ID type features (e.g., image [95, 98], audio [87, 96], video [20, 46], social network [27, 33],

Slide from Ce Zhang

How much does it cost…

…to train a 100 Trillion parameter model for 1 day on the cloud?

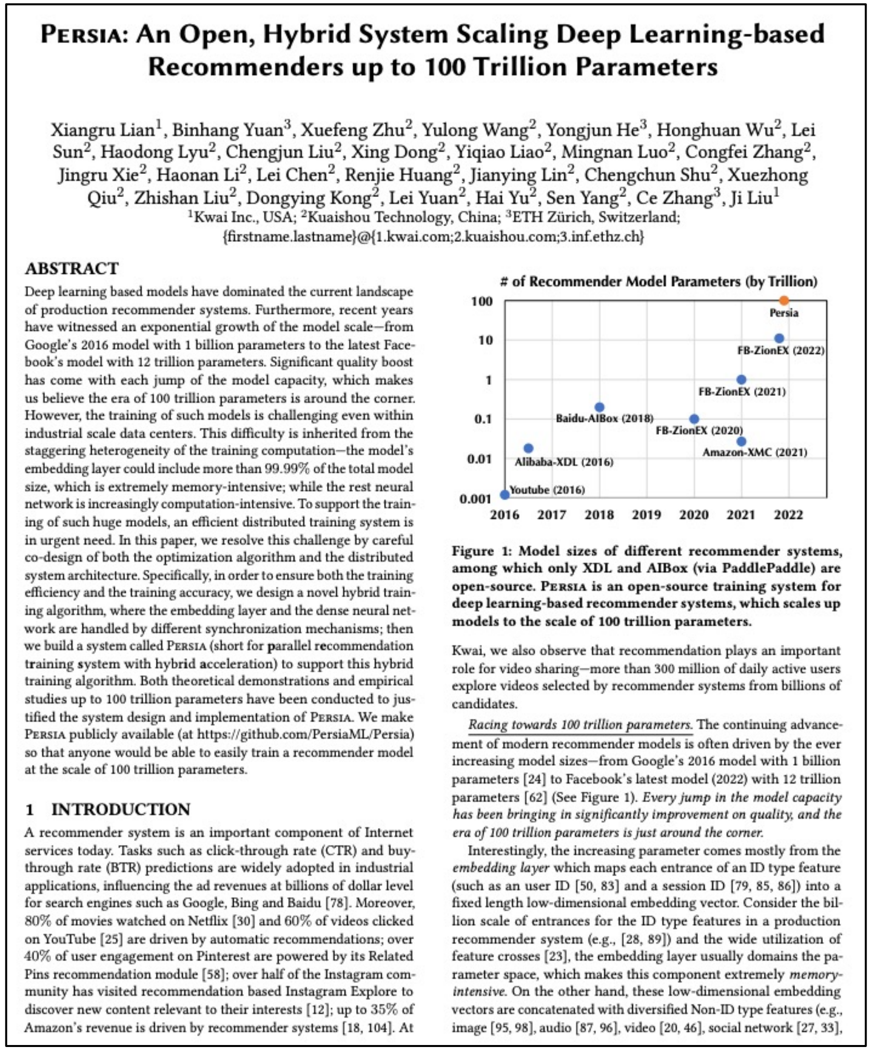
A. $4,000

B. $40,000

C. $400,000

Used:

- 3,000 CPU cores
- 64 A100 GPUs
- 360 TB of RAM



PERSIA: An Open, Hybrid System Scaling Deep Learning-based Recommenders up to 100 Trillion Parameters

Xiangru Lian[1], Binhang Yuan[3], Xuefeng Zhu[2], Yulong Wang[2], Yongjun He[3], Honghuan Wu[2], Lei Sun[2], Haodong Lyu[2], Chengjun Liu[2], Xing Dong[2], Yiqiao Liao[2], Mingnan Luo[2], Congfei Zhang[2], Jingru Xie[2], Haonan Li[2], Lei Chen[2], Renjie Huang[2], Jianying Lin[2], Chengchun Shu[2], Xuezhong Qiu[2], Zhishan Liu[2], Dongying Kong[2], Lei Yuan[2], Hai Yu[2], Sen Yang[2], Ce Zhang[3], Ji Liu[1]
[1]Kwai Inc., USA; [2]Kuaishou Technology, China; [3]ETH Zürich, Switzerland;
{firstname.lastname}@{1.kwai.com;2.kuaishou.com;3.inf.ethz.ch}

ABSTRACT

Deep learning based models have dominated the current landscape of production recommender systems. Furthermore, recent years have witnessed an exponential growth of the model scale—from Google's 2016 model with 1 billion parameters to the latest Facebook's model with 12 trillion parameters. Significant quality boost has come with each jump of the model capacity, which makes us believe the era of 100 trillion parameters is around the corner. However, the training of such models is challenging even within industrial scale data centers. This difficulty is inherited from the staggering heterogeneity of the training computation—the model's embedding layer could include more than 99.99% of the total model size, which is extremely memory-intensive; while the rest neural network is increasingly computation-intensive. To support the training of such huge models, an efficient distributed training system is in urgent need. In this paper, we resolve this challenge by careful co-design of both the optimization algorithm and the distributed system architecture. Specifically, in order to ensure both the training efficiency and the training accuracy, we design a novel hybrid training algorithm, where the embedding layer and the dense neural network are handled by different synchronization mechanisms; then we build a system called PERSIA (short for **p**arallel **re**commendation t**r**aining **s**ystem with hybr**i**d **a**cceleration) to support this hybrid training algorithm. Both theoretical demonstrations and empirical studies up to 100 trillion parameters have been conducted to justified the system design and implementation of PERSIA. We make PERSIA publicly available (at https://github.com/PersiaML/Persia) so that anyone would be able to easily train a recommender model at the scale of 100 trillion parameters.

1 INTRODUCTION

A recommender system is an important component of Internet services today. Tasks such as click-through rate (CTR) and buy-through rate (BTR) predictions are widely adopted in industrial applications, influencing the ad revenues at billions of dollar level for search engines such as Google, Bing and Baidu [78]. Moreover, 80% of movies watched on Netflix [30] and 60% of videos clicked on YouTube [25] are driven by automatic recommendations; over 40% of user engagement on Pinterest are powered by its Related Pins recommendation module [58]; over half of the Instagram community has visited recommendation based Instagram Explore to discover new content relevant to their interests [12]; up to 35% of Amazon's revenue is driven by recommender systems [18, 104]. At

Figure 1: Model sizes of different recommender systems, among which only XDL and AIBox (via PaddlePaddle) are open-source. PERSIA is an open-source training system for deep learning-based recommender systems, which scales up models to the scale of 100 trillion parameters.

Kwai, we also observe that recommendation plays an important role for video sharing—more than 300 million of daily active users explore videos selected by recommender systems from billions of candidates.

Racing towards 100 trillion parameters. The continuing advancement of modern recommender models is often driven by the ever increasing model sizes—from Google's 2016 model with 1 billion parameters [24] to Facebook's latest model (2022) with 12 trillion parameters [62] (See Figure 1). *Every jump in the model capacity has been bringing in significantly improvement on quality, and the era of 100 trillion parameters is just around the corner.*

Interestingly, the increasing parameter comes mostly from the *embedding layer* which maps each entrance of an ID type feature (such as an user ID [50, 83] and a session ID [79, 85, 86]) into a fixed length low-dimensional embedding vector. Consider the billion scale of entrances for the ID type features in a production recommender system (e.g., [28, 89]) and the wide utilization of feature crosses [23], the embedding layer usually domains the parameter space, which makes this component extremely *memory-intensive*. On the other hand, these low-dimensional embedding vectors are concatenated with diversified Non-ID type features (e.g., image [95, 98], audio [87, 96], video [20, 46], social network [27, 33],

Slide from Ce Zhang

Blog: https://cloud.google.com/blog/products/ai-machine-learning/training-a-recommender-model-of-100-trillions-parameters-on-google-cloud

ML has a cost & resource efficiency problem

ML has a cost & resource efficiency problem

- Another example:

https://www.mosaicml.com/blog/gpt-3-quality-for-500k

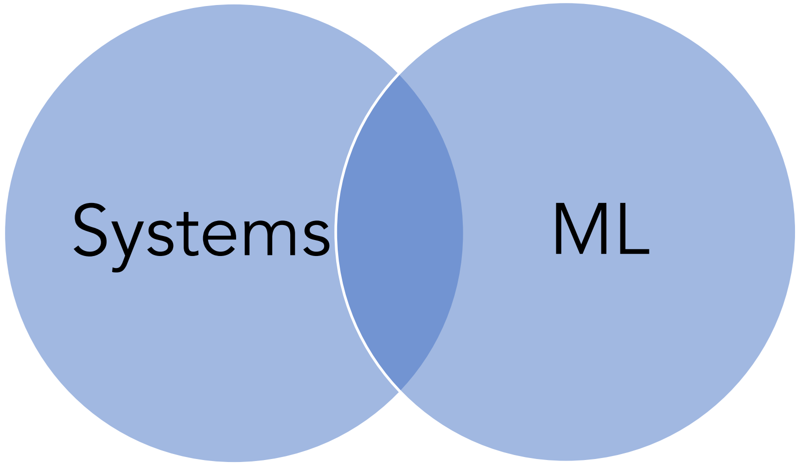
How can we reduce the cost of ML?

How can we reduce the cost of ML?

Many complementary approaches…

Improve:

- Resource efficiency
- Resource cost
- Model efficiency
- Data efficiency
- …



How can we reduce the cost of ML?

Many complementary approaches…

Improve:

- Resource efficiency → *maximize ML hardware (GPU/TPU) utilization*
- Resource cost
- Model efficiency
- Data efficiency
- …

How can we reduce the cost of ML?

Many complementary approaches…

Improve:

- Resource efficiency → *maximize ML hardware (GPU/TPU) utilization*
- Resource cost → *use cheap, transient resources (e.g., spot VMs)*
- Model efficiency
- Data efficiency
- …

How can we reduce the cost of ML?

Many complementary approaches…

Improve:

- Resource efficiency → *maximize ML hardware (GPU/TPU) utilization*
- Resource cost → *use cheap, transient resources (e.g., spot VMs)*
- Model efficiency → *sparsely activate models, sparse architectures*
- Data efficiency
- …

How can we reduce the cost of ML?

Many complementary approaches…

Improve:

- Resource efficiency → *maximize ML hardware (GPU/TPU) utilization*
- Resource cost → *use cheap, transient resources (e.g., spot VMs)*
- Model efficiency → *sparsely activate models, sparse architectures*
- Data efficiency → *train on the most important/relevant data*
- …

How can we reduce the cost of ML?

Many complementary approaches…

Improve:

- Resource efficiency → *maximize ML hardware (GPU/TPU) utilization*

If we can ensure a job makes "good use" of ML hardware, the job will finish faster and we will pay for less time on that hardware.

- …

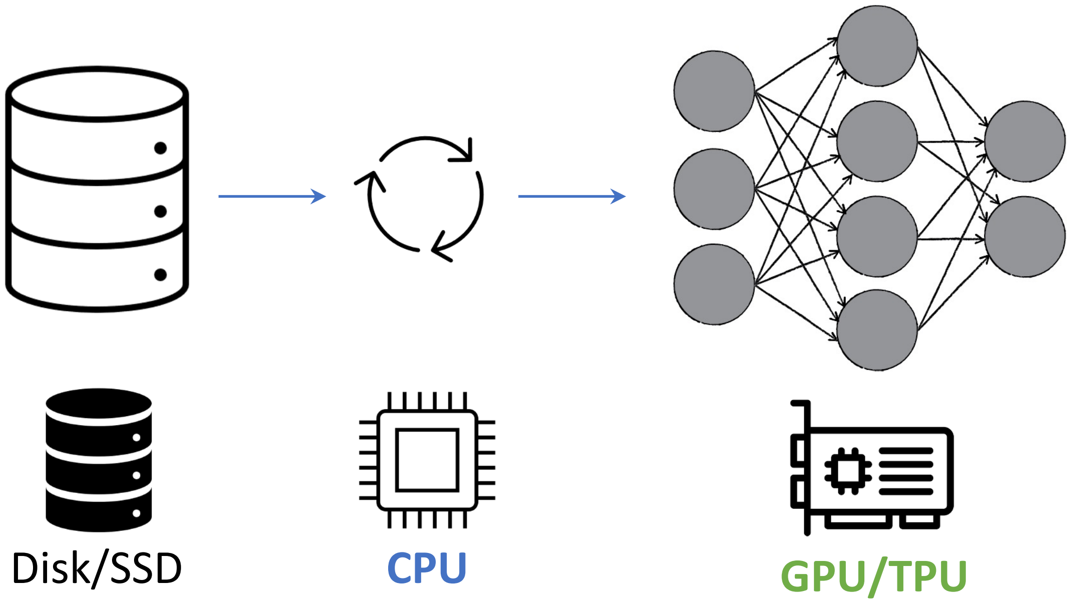
Focus on maximizing **GPU/TPU** utilization → most $$$ component

What hinders high GPU/TPU utilization?

- Feeding GPUs/TPUs with input data is often a bottleneck

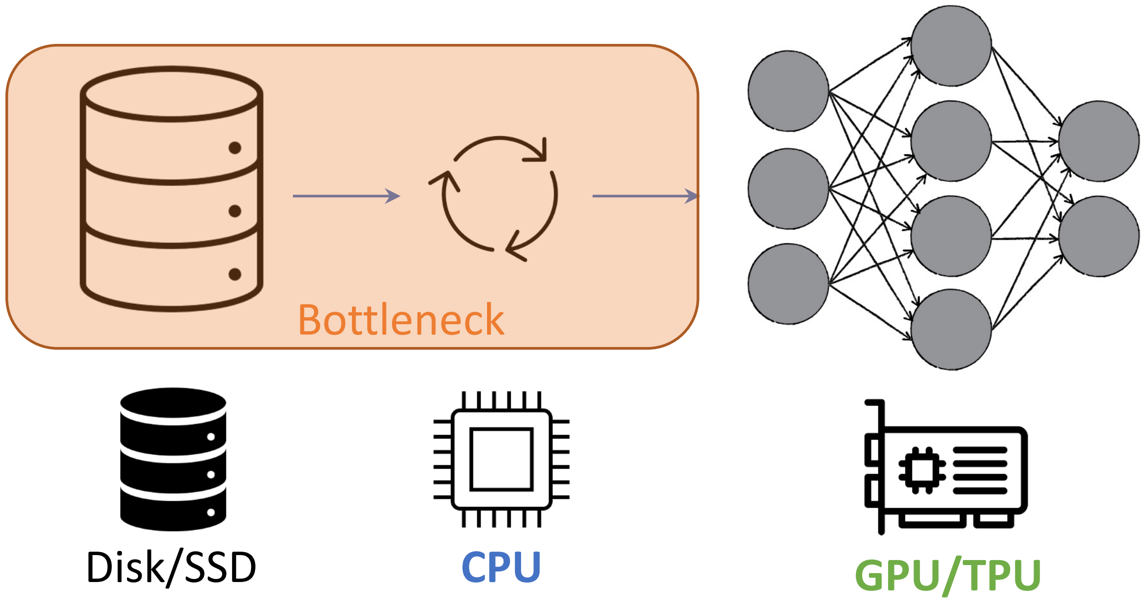
What hinders high GPU/TPU utilization?

- Feeding GPUs/TPUs with input data is often a bottleneck
  - Need to read large volumes of data from storage and preprocess data



What hinders high GPU/TPU utilization?

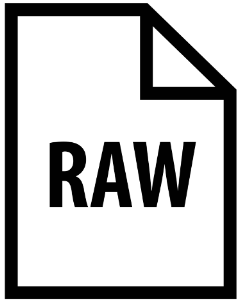
- Feeding GPUs/TPUs with input data is often a bottleneck
  - Need to read large volumes of data from storage and preprocess data



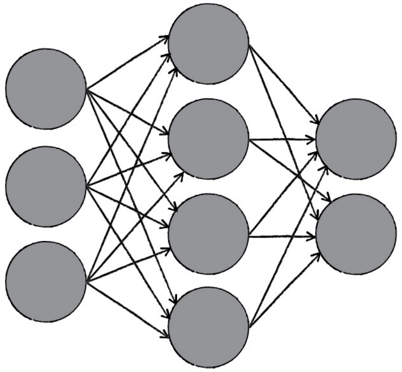
Input data ingestion for ML

Before we can feed training data to a model, we need to preprocess data.

Raw Data

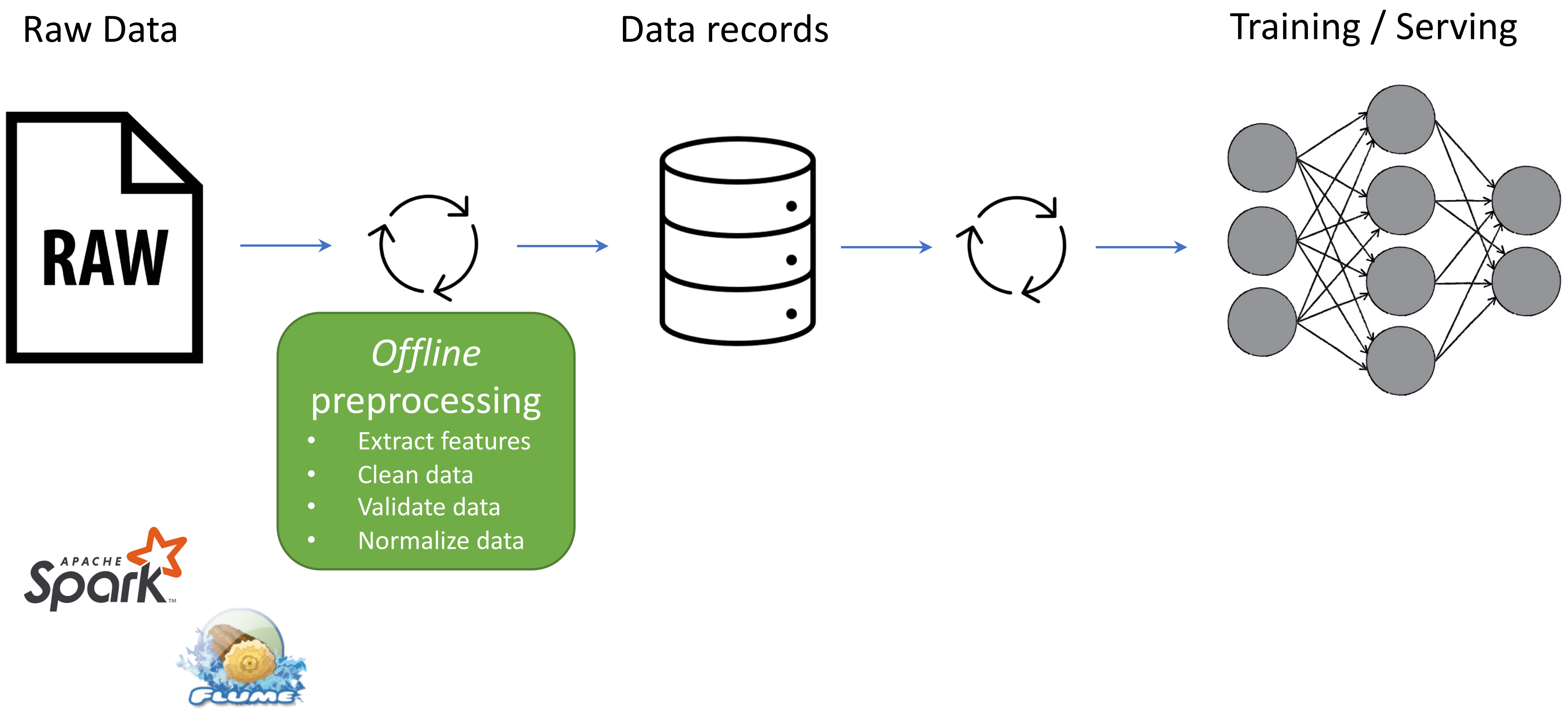


Training / Serving



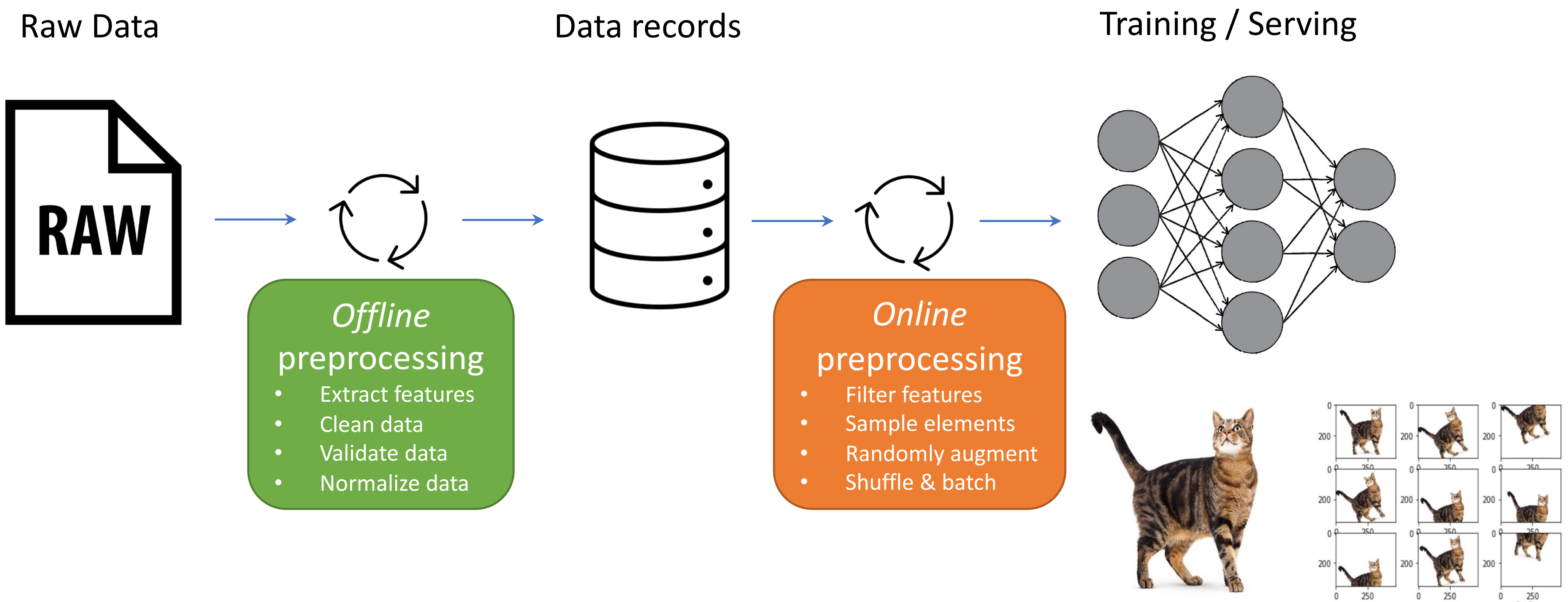
Input data processing for ML

Before we can feed training data to a model, we need to preprocess data.



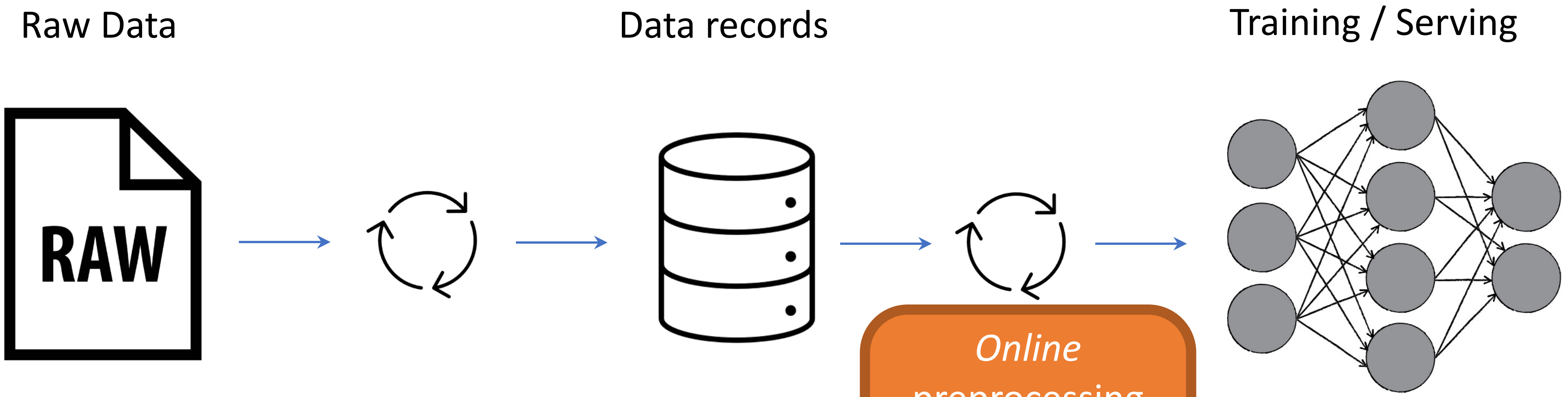
Input data processing for ML

Before we can feed training data to a model, we need to preprocess data.



Input data processing for ML

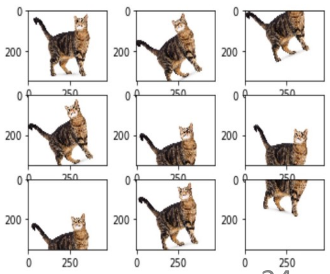
Before we can feed training data to a model, we need to preprocess data.



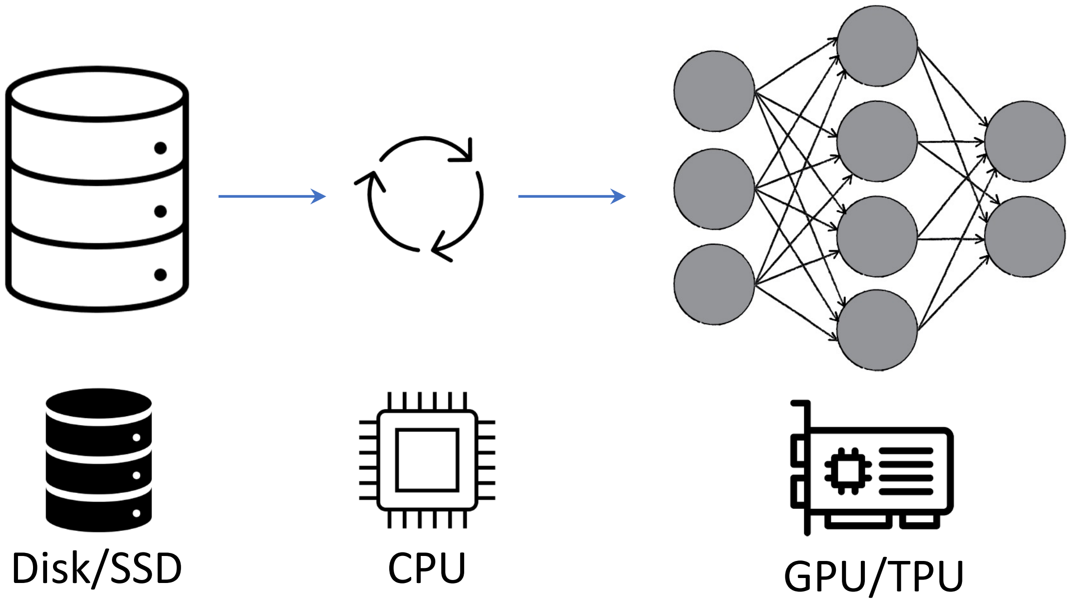
Online preprocessing

- Filter features
- Sample elements
- Randomly augment
- Shuffle & batch

"Last mile"
data processing

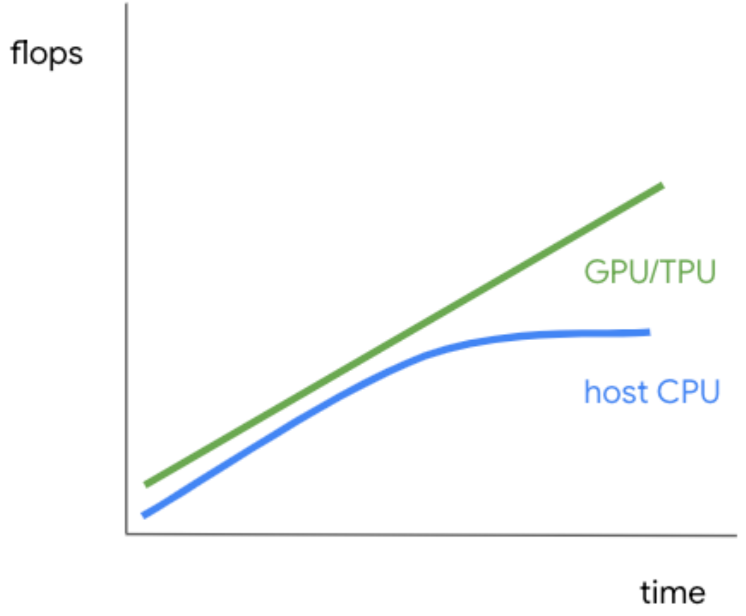


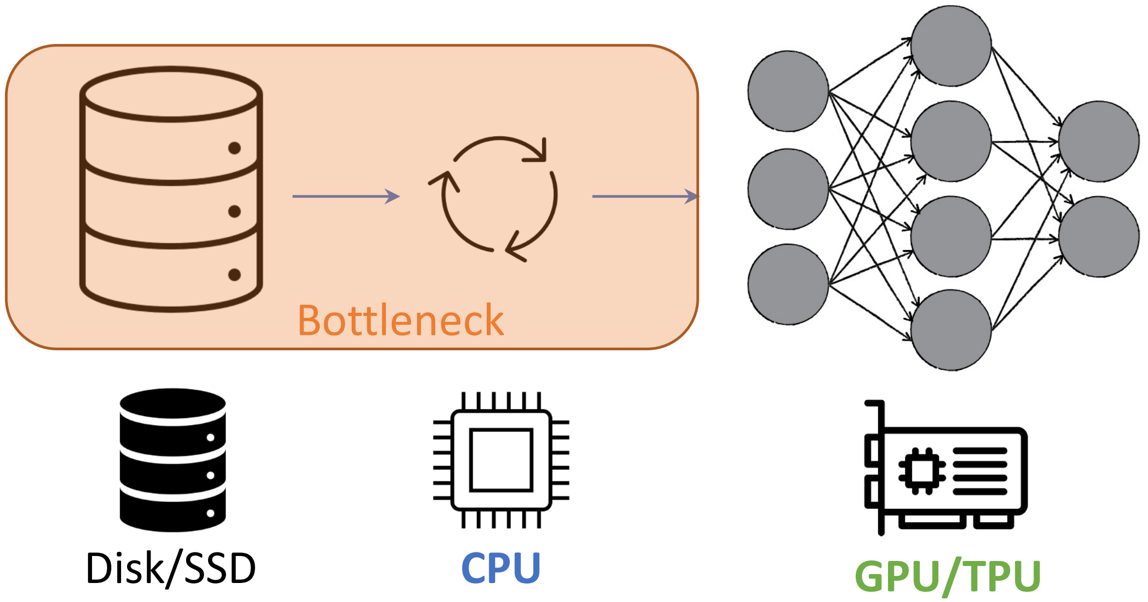
Input processing impacts training time & cost



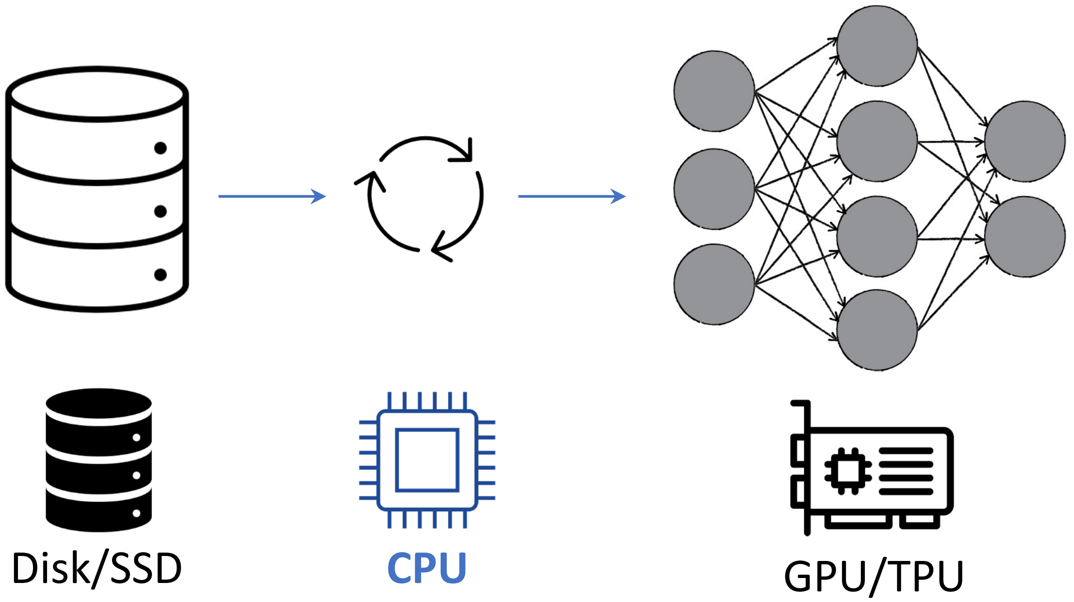
Input processing impacts training time & cost

- Feeding data-hungry GPUs/TPUs is challenging
  - Input data processing on host CPU is often a bottleneck



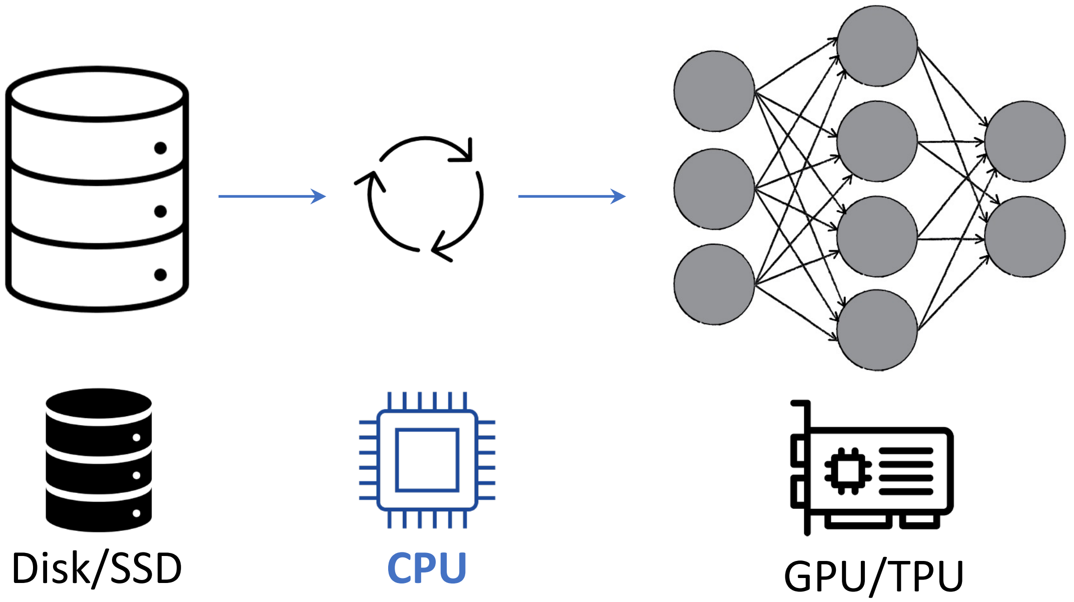


Input processing consumes high CPU/energy



Input processing consumes high CPU/energy

- At Google, data processing consumes ~30% of compute time in training jobs [1]
- At Meta, data processing consumes more power than training for some jobs [2]



[1] Derek G. Murray, Jiří Šimša, Ana Klimovic, Ihor Indyk: "tf.data: A Machine Learning Data Processing Framework". VLDB 2021.

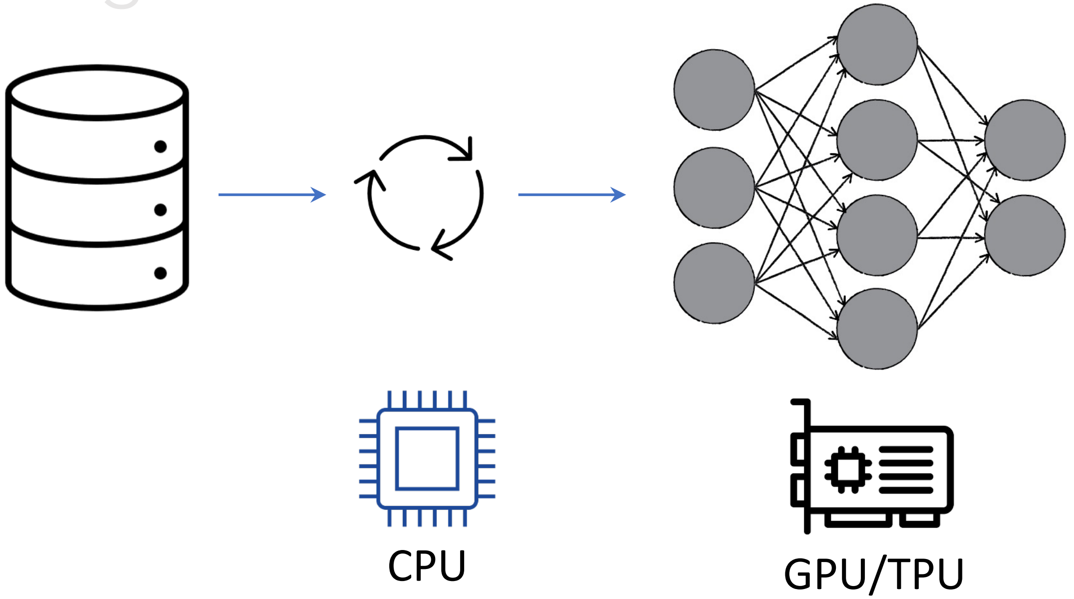
[2] Mark Zhao et al. "Understanding data storage and ingestion for large-scale deep recommendation model training", ISCA 2022.

How to optimize ML input data processing?

1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service

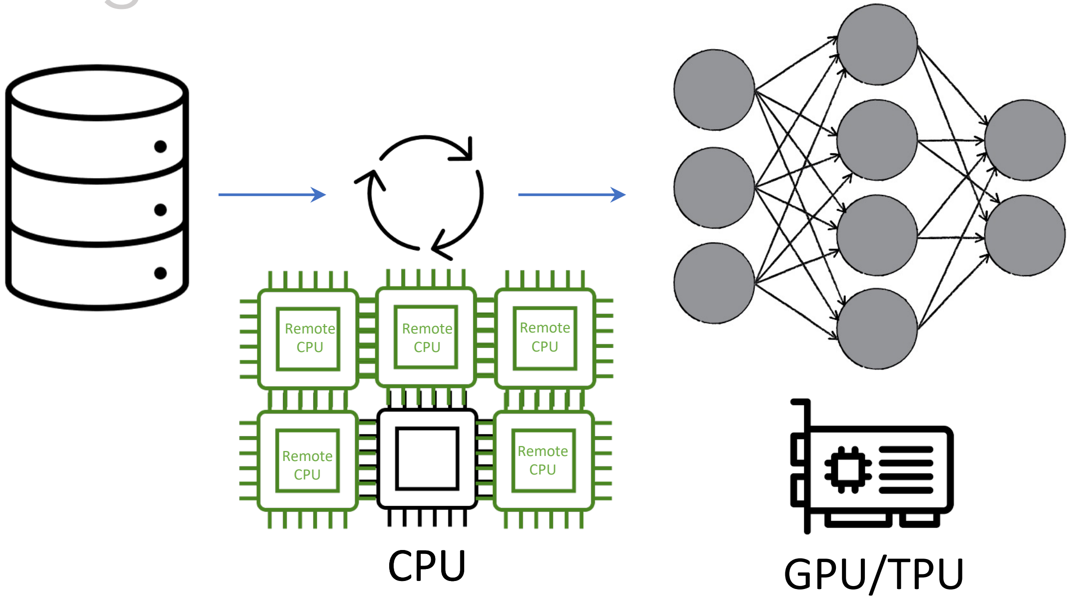
How to optimize ML input data processing?

1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service



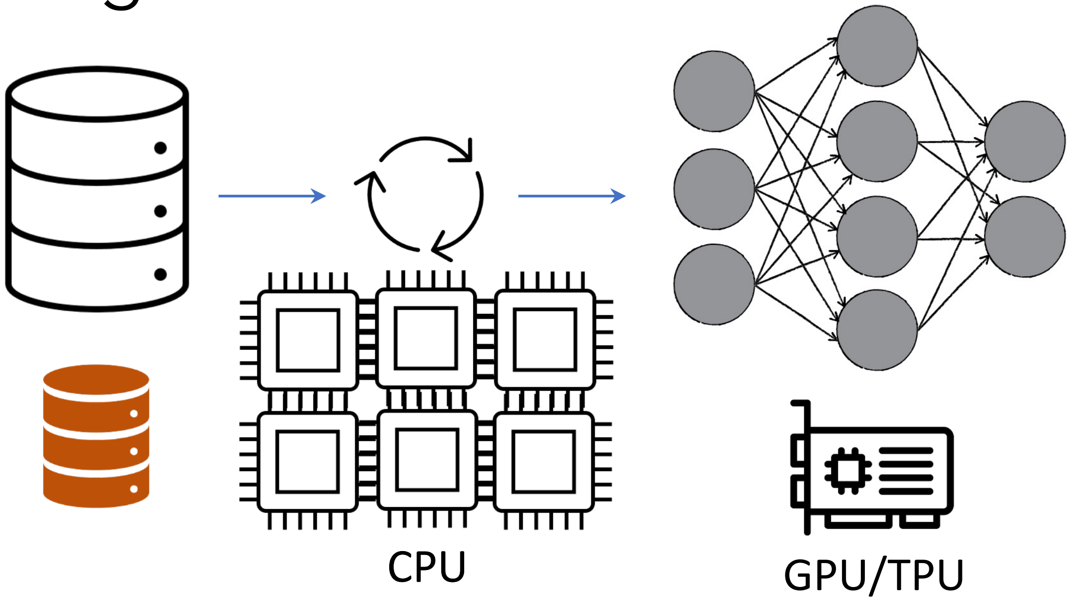
How to optimize ML input data processing?

1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service



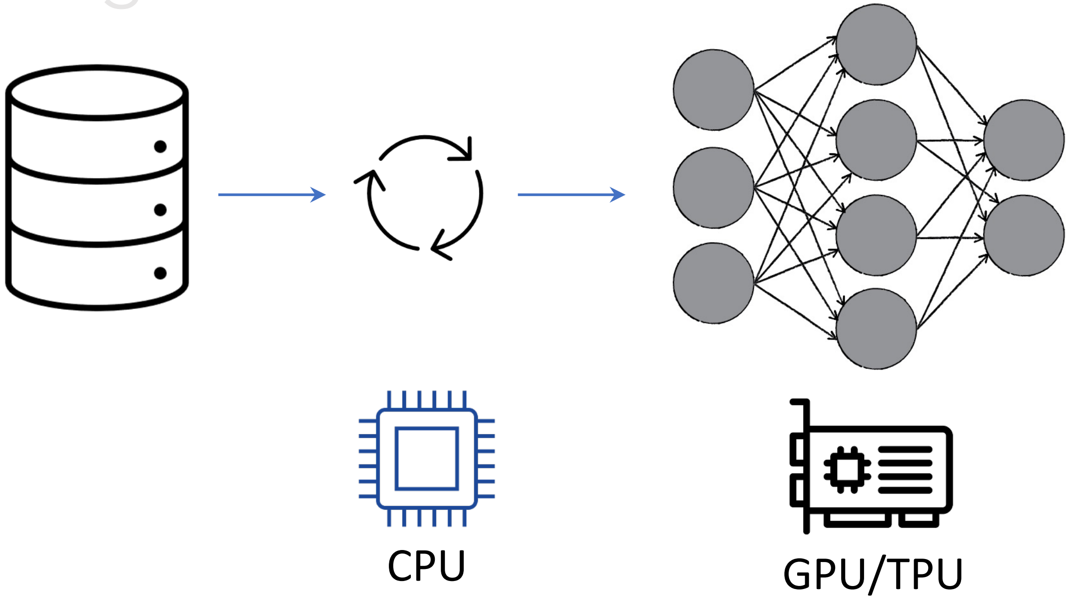
How to optimize ML input data processing?

1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service



How to optimize ML input data processing?

1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service



tf.data: ML input data processing framework

- API provides generic operators that can be composed & parameterized:
  - Consists of stateless *datasets* (to define pipeline) and stateful *iterators* (to produce elements)

Derek G. Murray, Jiří Šimša, Ana Klimovic, Ihor Indyk: "tf.data: A Machine Learning Data Processing Framework". VLDB 2021.

tf.data: ML input data processing framework

- API provides generic operators that can be composed & parameterized:
  - Consists of stateless *datasets* (to define pipeline) and stateful *iterators* (to produce elements)

`read(file)` → `map(parse)` → `filter(cond)` → `map(crop)` → `shuffle()` → `batch()` → `prefetch()`

Derek G. Murray, Jiří Šimša, Ana Klimovic, Ihor Indyk: "tf.data: A Machine Learning Data Processing Framework". VLDB 2021.

tf.data: ML input data processing framework

- API provides generic operators that can be composed & parameterized:
  - Consists of stateless *datasets* (to define pipeline) and stateful *iterators* (to produce elements)

`read(file)` → `map(parse)` → `filter(cond)` → `map(crop)` → `shuffle()` → `batch()` → `prefetch()`

- Runtime efficiently executes input pipelines by applying:
  - Software pipelining and parallelism
  - Static optimizations (e.g., operator fusion)
  - Dynamic optimizations (autotuning parallelism & prefetch buffer sizes)

Derek G. Murray, Jiří Šimša, Ana Klimovic, Ihor Indyk: "tf.data: A Machine Learning Data Processing Framework". VLDB 2021.

```python
import tensorflow as tf

def preprocess(record):
  ...


dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)


model = ...
model.fit(dataset, epochs=10)
```

```python
import tensorflow as tf

def preprocess(record):
  ...


dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)


model = ...
model.fit(dataset, epochs=10)
```

read data from storage

```python
import tensorflow as tf

def preprocess(record):
  ...


dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)


model = ...
model.fit(dataset, epochs=10)
```

apply user-defined preprocessing

```python
import tensorflow as tf

def preprocess(record):
  ...


dataset = tf.data.TFRec
dataset = dataset.map(p eprocess)
dataset = dataset.batch(batch_size=32)


model = ...
model.fit(dataset, epochs=10)
```

batch data for training efficiency

```python
import tensorflow as tf

def preprocess(record):
  ...


dataset = tf.data.T
dataset = dataset.ma
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch(buffer_size=X)

model = ...
model.fit(dataset, epochs=10)
```

overlap data processing and loading

```python
import tensorflow as tf

def preprocess(record):
  ...


dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch(buffer_size=X)

model = ...
model.fit(dataset, epochs=10)
```

train model with tf.data dataset

```python
import tensorflow as tf

def preprocess(record):
  ...


dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch(buffer_size=X)

model = ...
model.fit(dataset, epochs=10)
```

tf.data runtime applies optimizations to the input pipeline under the hood

Example of optimization:
`map+batch` fusion

```python
import tensorflow as tf

def preprocess(record):
  ...


dataset = tf.data.TFRecordDataset(".../*.tfrecord", num_parallel_readers=Z)
dataset = dataset.map(preprocess, num_parallel_calls=Y)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch(buffer_size=X)

model = ...
model.fit(dataset, epochs=10)
```

tf.data runtime applies optimizations to the input pipeline under the hood

Software parallelism & pipelining

```python
import tensorflow as tf

def preprocess(record):
  ...
```

tf.data runtime applies optimizations to the input pipeline under the hood

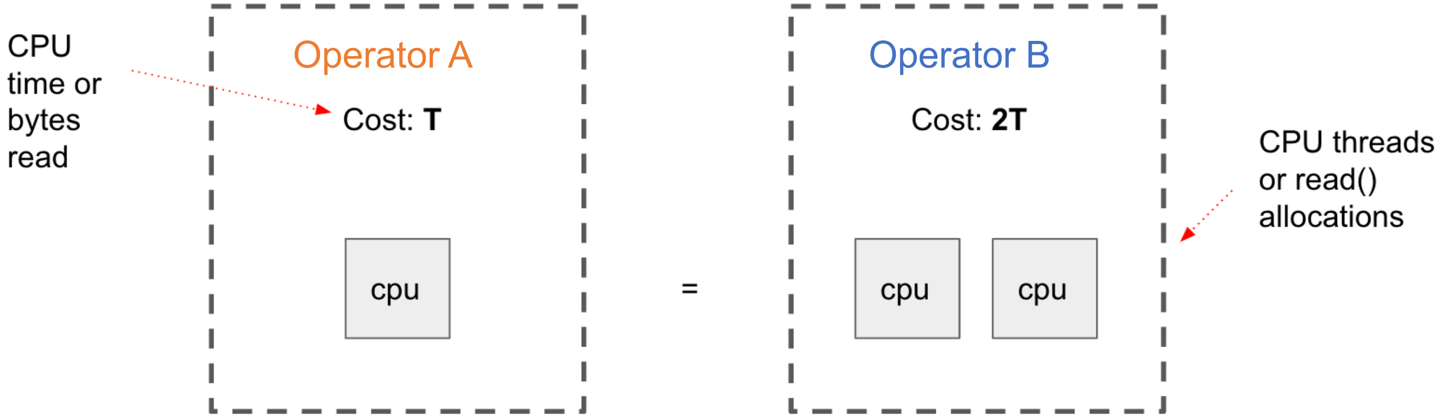
```python
dataset = tf.data.TFRecordDataset(".../*.tfrecord", num_parallel_readers=Z)
dataset = dataset.map(preprocess, num_parallel_calls=Y)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch(buffer_size=X)

model = ...
model.fit(dataset, epochs=10)
```

`tf.data.AUTOTUNE`

Hill-climbing algorithm tunes CPU/mem allocations to minimize output latency, modelled by M/M/1/k queue at each iterator

Plumber: input pipeline perf debug/tuning

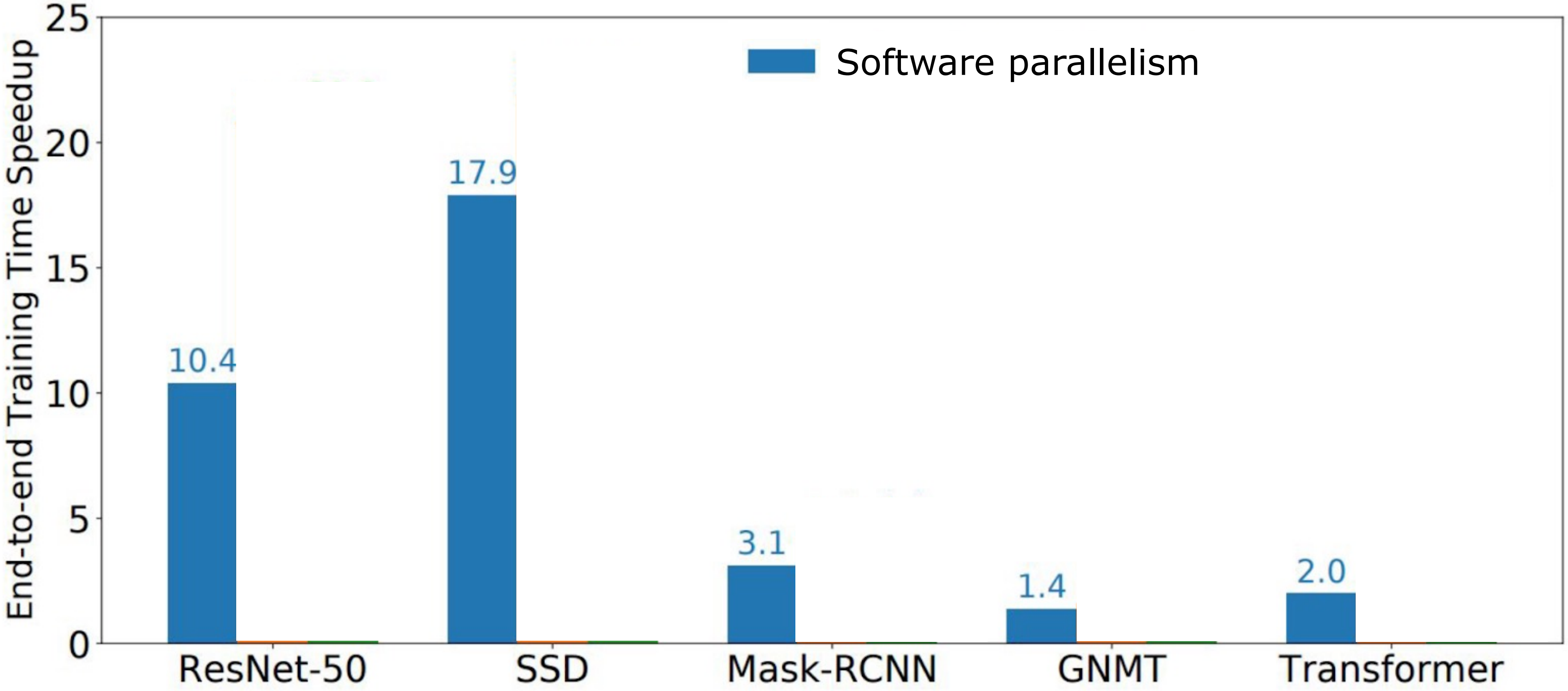
- Identify which op of the input pipeline is the bottleneck
- Adjust op CPU/memory/storage resource allocations to alleviate bottlenecks:
  - Measure resource accounted rate (i.e., "cost") for each operator
  - If Operator B is twice as "expensive" as Operator A, give Operator B twice the resources
  - Cast resource allocation as an integer linear programming problem



Michael Kuchnik et al. *Plumber: Diagnosing and Removing Performance Bottlenecks in Machine Learning Data Pipelines*. MLSys'22.

Training speedup with tf.data optimizations

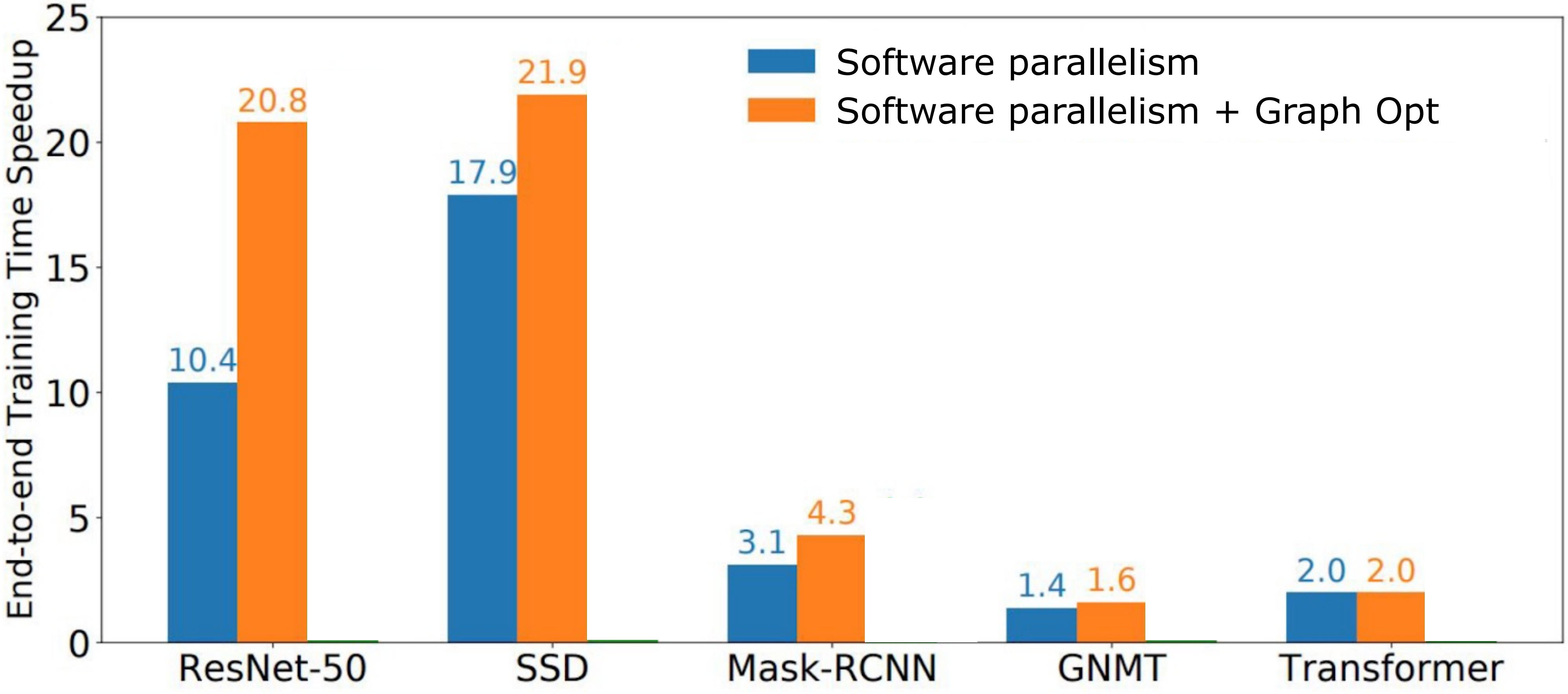
Baseline is input pipeline logic with no software parallelism or graph optimizations.



Derek G. Murray, Jiří Šimša, Ana Klimovic, Ihor Indyk. *tf.data: A Machine Learning Data Processing Framework*. VLDB 2021.

Training speedup with tf.data optimizations

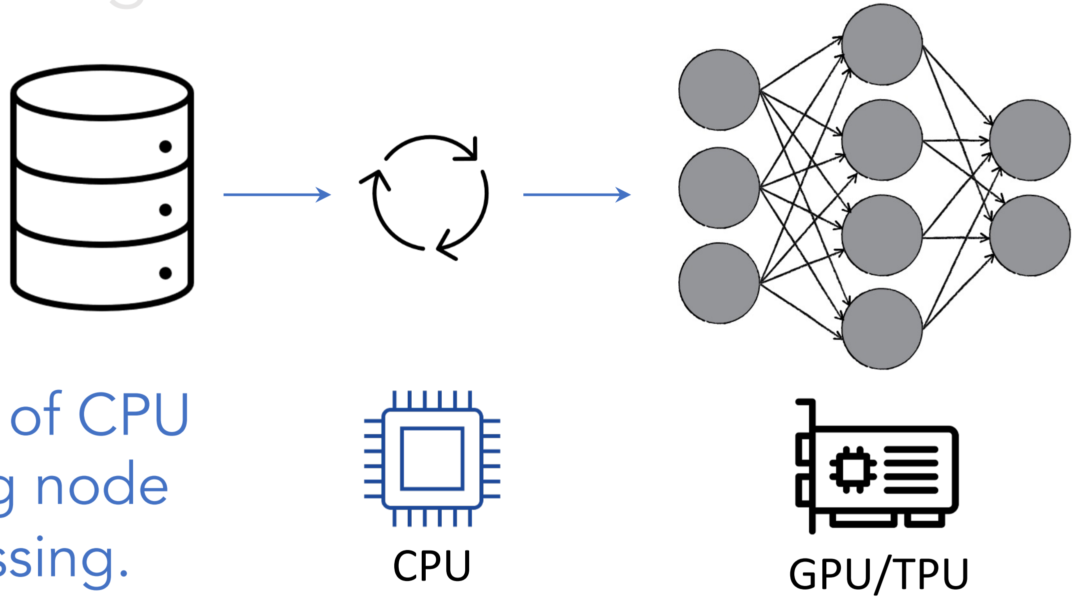
Baseline is input pipeline logic with no software parallelism or graph optimizations.



Derek G. Murray, Jiří Šimša, Ana Klimovic, Ihor Indyk. ***tf.data: A Machine Learning Data Processing Framework***. VLDB 2021.

How to optimize ML input data processing?

1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service

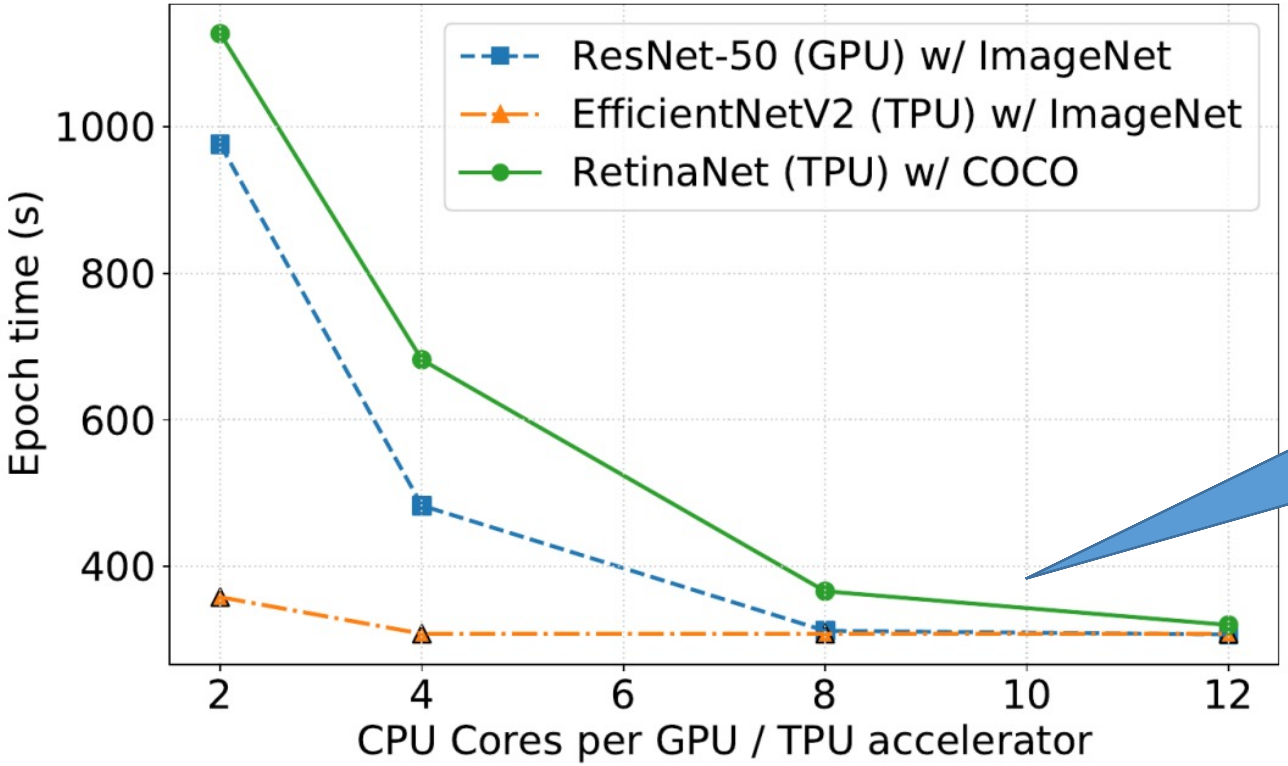


Autotuning tries to make best use of CPU and RAM available on the training node for high-throughput data processing.

How much CPU/RAM to provision per GPU/TPU?

It is hard to determine the right resource ratio for a ML training node.

→ Ideal resource allocation depends on the model and input pipeline

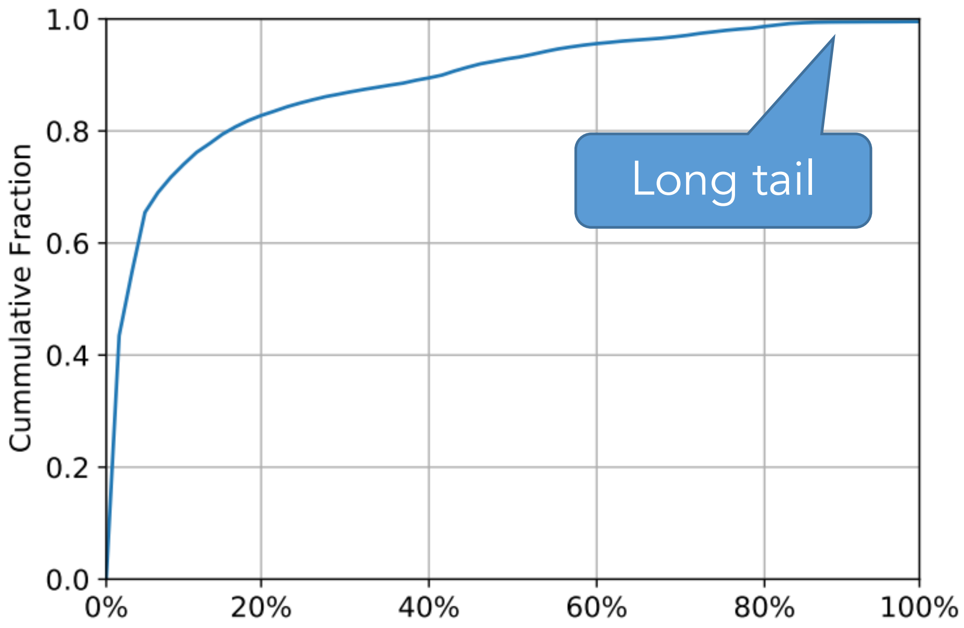


Training jobs benefit differently when given more CPU for data processing per accelerator core

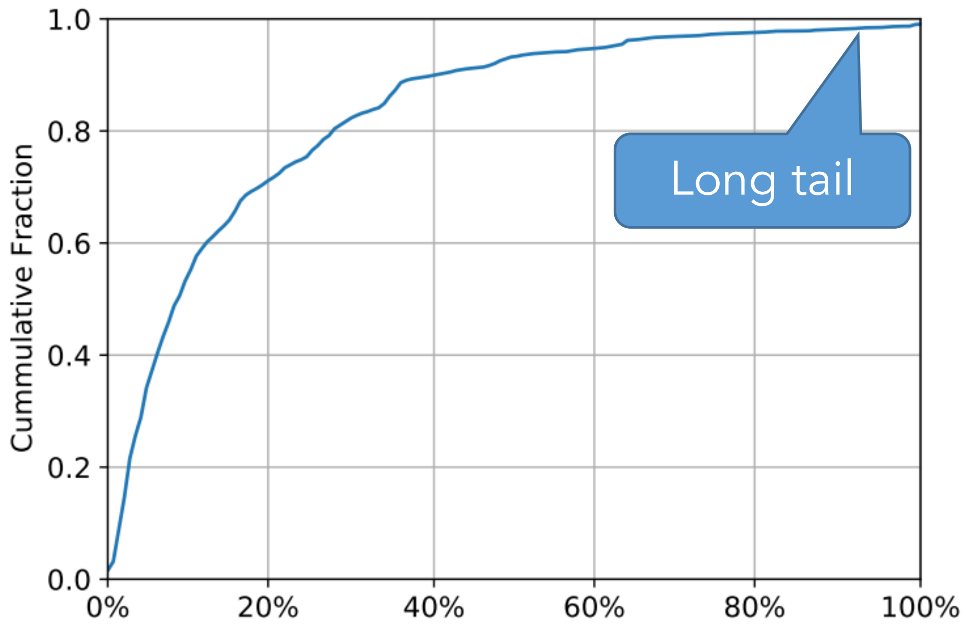
How much CPU/RAM to provision per GPU/TPU?

It is hard to determine the right resource ratio for a ML training node.

Example of normalized CPU and RAM usage CDF, from ~73K ML training jobs at Google:



(a) CPU usage



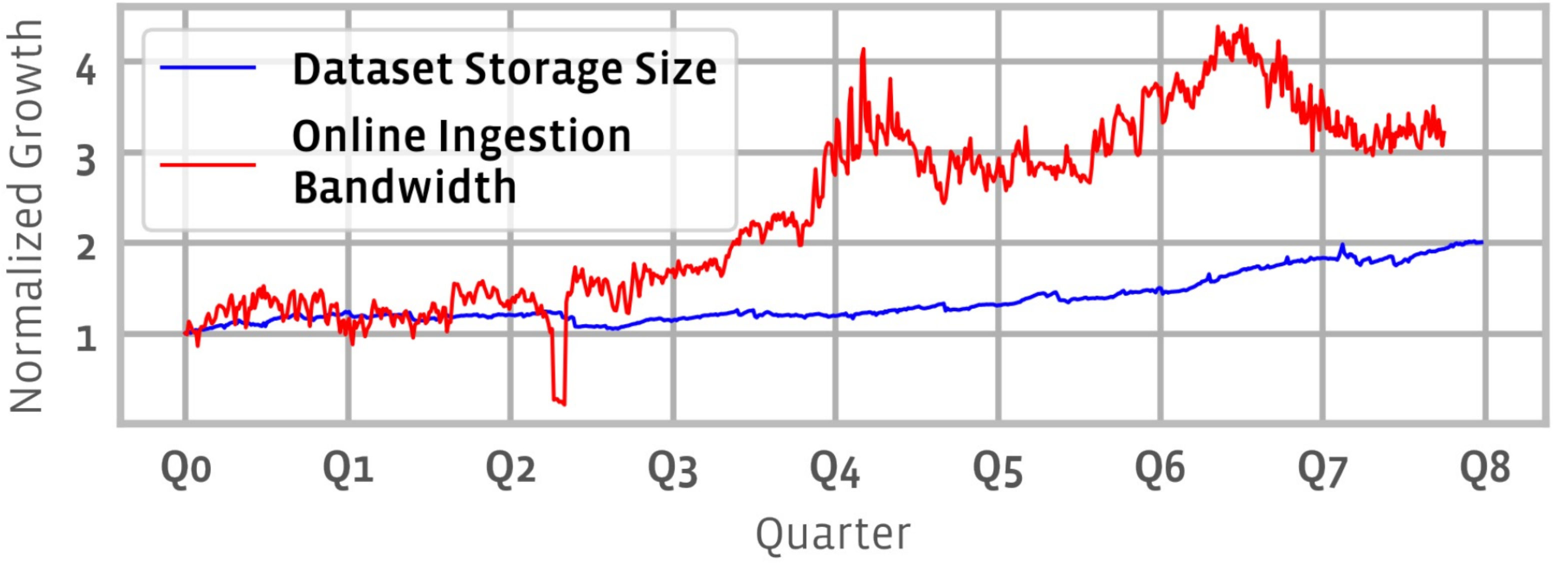
(b) RAM usage

Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiri Simsa, Chandu Thekkath. *A case for disaggregation of ML data processing*, 2022.

We need a scalable data processing architecture

Need to adjust resource allocation over time. ML training is increasingly data-hungry.

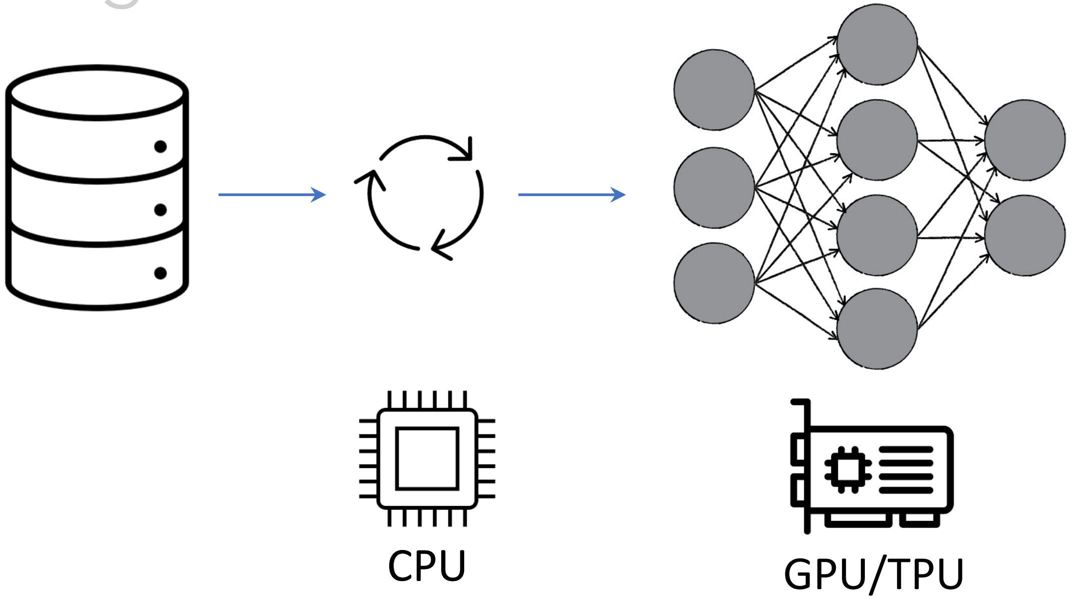
At Meta, storage and bandwidth has grown over 2x and 4x over the past 2 years.



Mark Zhao et al. "Understanding data storage and ingestion for large-scale deep recommendation model training", ISCA 2022.

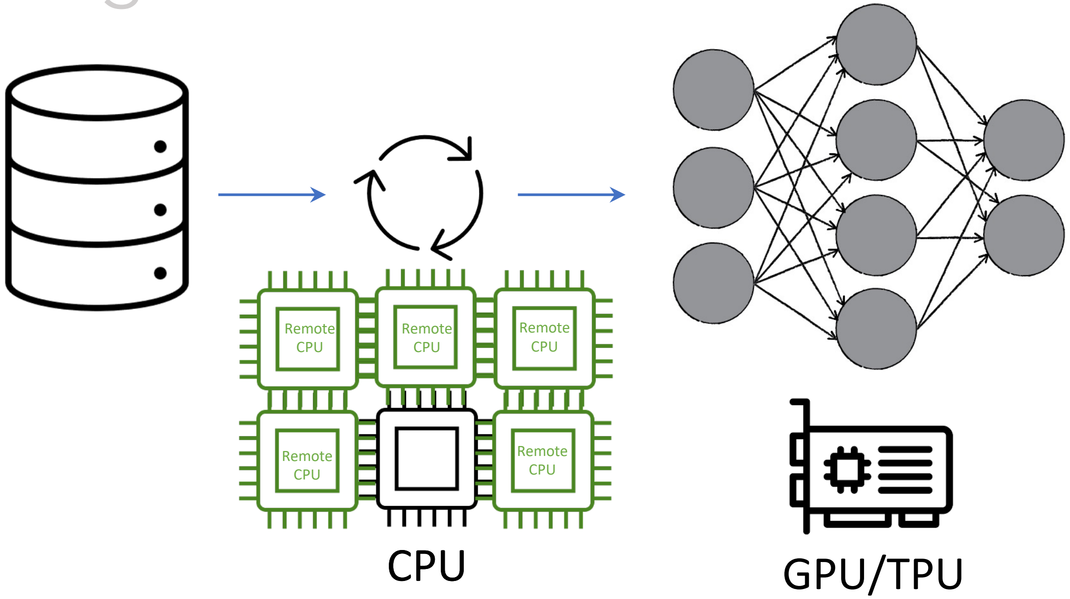
How to optimize ML input data processing?

1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service



How to optimize ML input data processing?

1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service



Solution: disaggregate data processing

- Independently scale resources for input data processing & model training

Solution: disaggregate data processing

- Independently scale resources for input data processing & model training
- Approach taken at Google (*tf.data service*), Meta (*DPP*), …

A case for disaggregation of ML data processing

Andrew Audibert, *Google*; Yang Chen, *Google*; Dan Graur, *ETH Zurich*; Ana Klimovic, *ETH Zurich*; Jiří Šimša, *Google*
Chandramohan A. Thekkath, *Google*

Abstract

Machine Learning (ML) computation requires feeding input data for the models to ingest. Traditionally, input data processing happens on the same host as the ML computation [8, 25]. The input data processing can however become a bottleneck of the ML computation if there are insufficient resources to process data quickly enough. This slows down the ML computation and wastes valuable and scarce ML hardware (e.g. GPUs and TPUs) used by the ML computation.

In this paper, we present *tf.data service*, a disaggregated input data processing service built on top of tf.data. Our work goes beyond describing the design and implementation of a new system which disaggregates preprocessing from ML computation and presents: (1) empirical evidence based on production workloads for the need of disaggregation, as well as quantitative evaluation of the impact disaggregation has on the performance and cost of production workloads, (2) benefits of disaggregation beyond horizontal scaling, (3) analysis of tf.data service's adoption at Google, the lessons learned during building and deploying the system and potential future lines of research opened up by our work.

We demonstrate that horizontally scaling data processing using tf.data service helps remove input bottlenecks, achieving speedups of up to 110× and job cost reductions of up to 89×. We further show that tf.data service can support computation reuse through data sharing across ML jobs with iden-

To enable high utilization of ML hardware, Google built and open-sourced the tf.data framework [25]. tf.data provides an efficient runtime to execute ML input data pipelines and a convenient API to express input data transformations. Since its launch in 2017, tf.data has grown in adoption to become the predominant solution for data ingestion and processing of ML computations at Google. All Google-based submissions to the ML Perf training competition [22] in recent years have relied on tf.data to achieve high performance. The framework is also widely used by open-source Tensorflow [1] programs.

However, tf.data could not meet the needs of all Tensorflow programs. The original design colocated data ingestion and processing with the ML computations. For some Tensorflow programs, host resources used for colocated data processing (CPU, RAM, and I/O bandwidth) became the bottleneck, leaving expensive ML hardware underutilized. This increases the end-to-end execution time and cost of ML jobs.

The fundamental challenge is that ML jobs have a wide spectrum of CPU and memory requirements, which make it impossible to right-size host CPU and memory resources (for data processing) colocated with specialized ML accelerators (for ML computations). Evidence of this is shown in Figure 1. By pre-provisioning colocated preprocessing resources, a one-size-fits-all resource deployment is imposed on ML preprocessing which is only optimal for a narrow subset of all potential ML jobs. Most jobs will either end up using a frac-

Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training

Industrial Product*

Mark Zhao†, Niket Agarwal†, Aarti Basant†, Buğra Gedik†, Satadru Pan†, Mustafa Ozdal†, Rakesh Komuravelli†, Jerry Pan†, Tianshu Bao†, Haowei Lu†, Sundaram Narayanan†, Jack Langman†, Kevin Wilfong†, Harsha Rastogi†, Carole-Jean Wu†, Christos Kozyrakis‡, Parik Pol†

†Meta, ‡Stanford University

ABSTRACT

Datacenter-scale AI training clusters consisting of thousands of domain-specific accelerators (DSA) are used to train increasingly-complex deep learning models. These clusters rely on a data storage and ingestion (DSI) pipeline, responsible for storing exabytes of training data and serving it at tens of terabytes per second. As DSAs continue to push training efficiency and throughput, the DSI pipeline is becoming the dominating factor that constrains the overall training performance and capacity. Innovations that improve the efficiency and performance of DSI systems and hardware are urgent, demanding a deep understanding of DSI characteristics and infrastructure at scale.

This paper presents Meta's end-to-end DSI pipeline, composed of a central data warehouse built on distributed storage and a Data PreProcessing Service that scales to eliminate data stalls. We characterize how hundreds of models are collaboratively trained across geo-distributed datacenters via diverse and continuous training jobs. These training jobs read and heavily filter massive and evolving datasets, resulting in popular features and samples used across training jobs. We measure the intense network, memory, and compute resources required by each training job to preprocess samples during training. Finally, we synthesize key takeaways based on our production infrastructure characterization. These include identifying hardware bottlenecks, discussing opportunities for heterogeneous DSI hardware, motivating research in datacenter scheduling and benchmark datasets, and assimilating lessons learned in optimizing DSI infrastructure.

KEYWORDS

Machine learning systems, databases, distributed systems, data ingestion, data storage

ACM Reference Format:

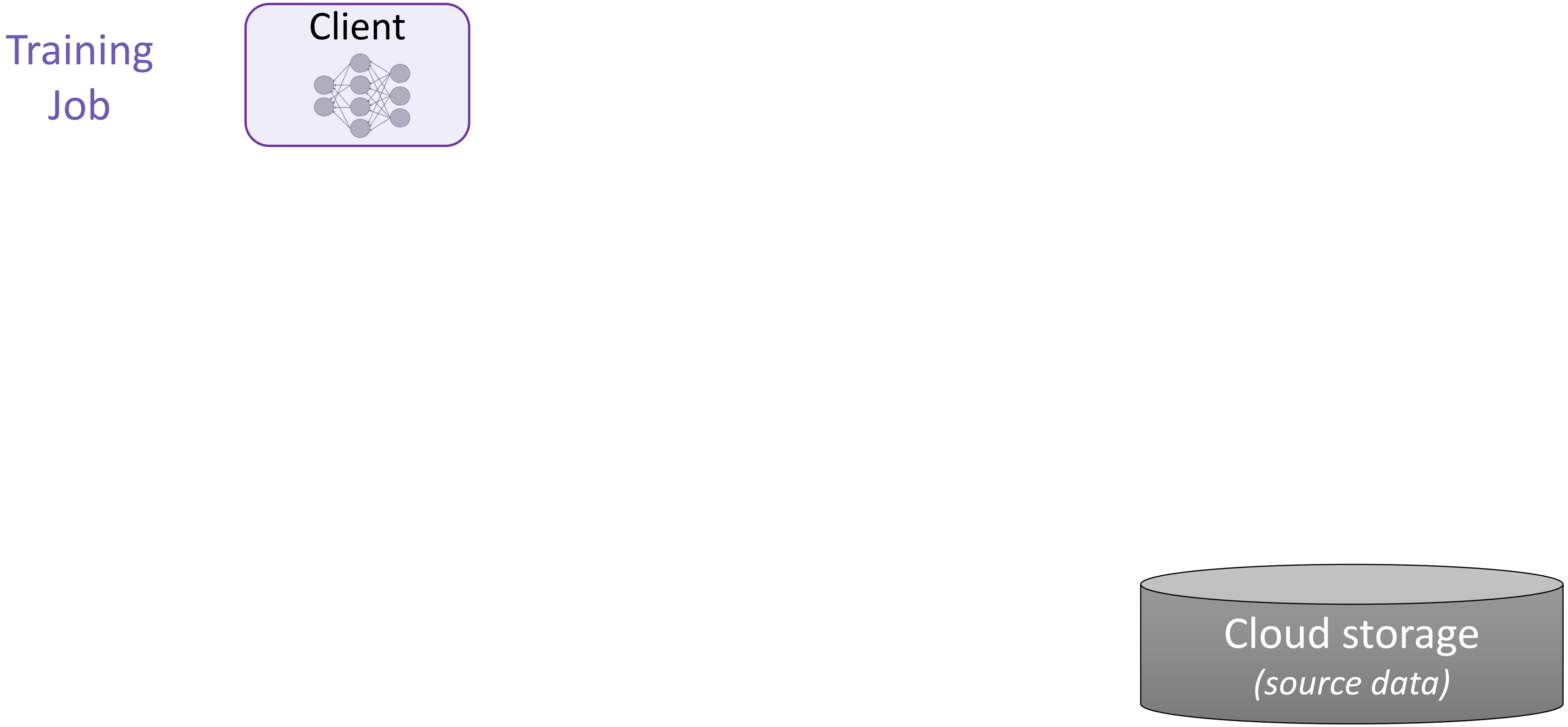
Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, Parik Pol. 2022. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3470496.3533044

1 INTRODUCTION

Domain-specific accelerators (DSAs) for deep neural networks (DNNs) have become ubiquitous because of their superior performance per watt over traditional general purpose processors [40]. Industry has rapidly embraced DSAs for both DNN training and inference. These DSAs include both traditional technologies, such as GPUs and FPGAs, as well as application-specific integrated circuits (ASICs) from, e.g., Habana [37], Graphcore [45], SambaNova [67], Tenstorrent [74], Tesla [75], AWS [23], Google [40], and others.

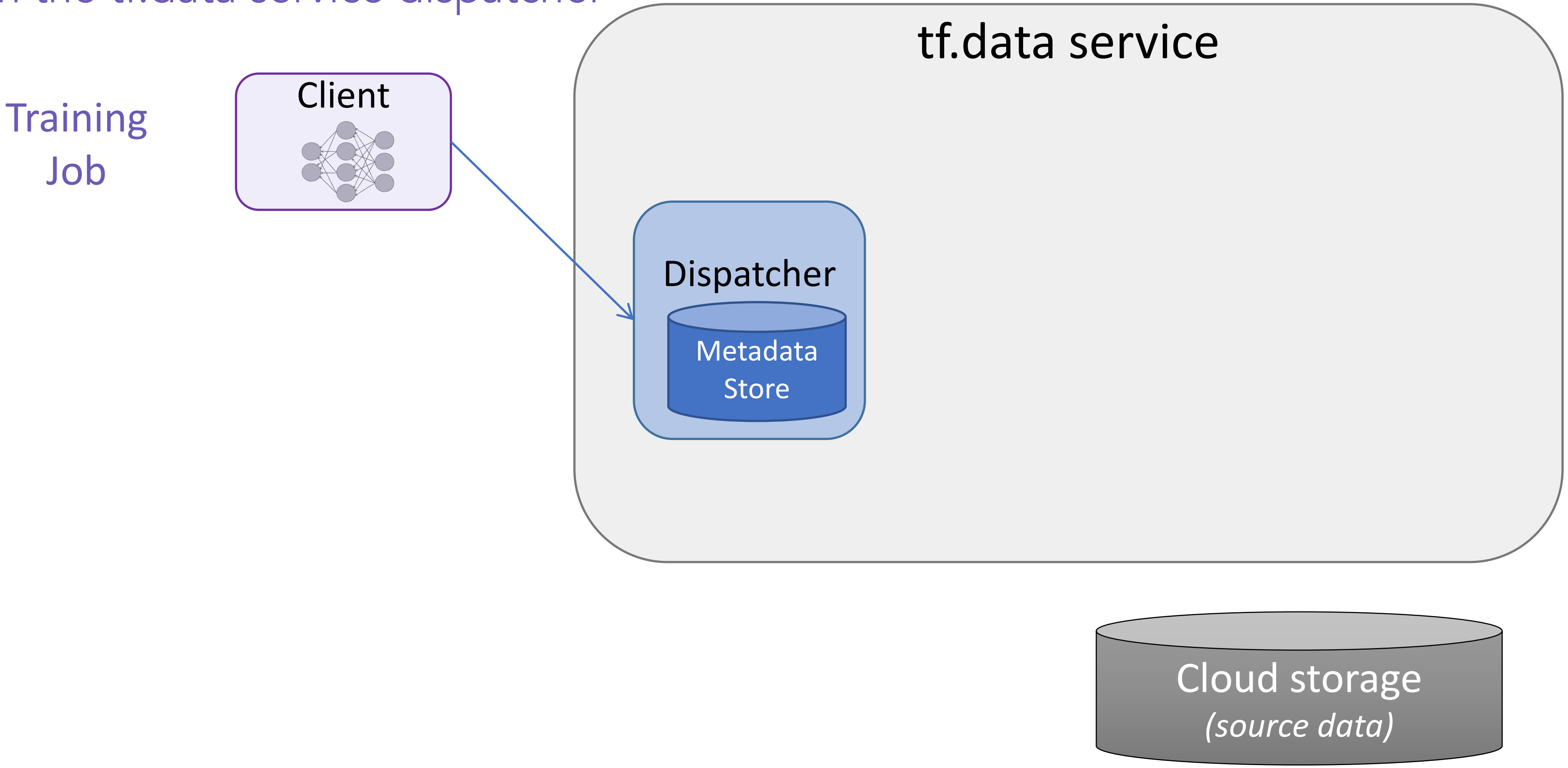
DSAs are increasingly deployed in immense scale-out systems to train increasingly-complex and computationally-demanding DNNs using massive datasets. For example, the latest MLPerf Training round (v1.1) [56] contains submissions from Azure and NVIDIA using 2048 and 4320 A100 GPUs, respectively, whereas Google submit-

tf.data service: disagg ML data processing



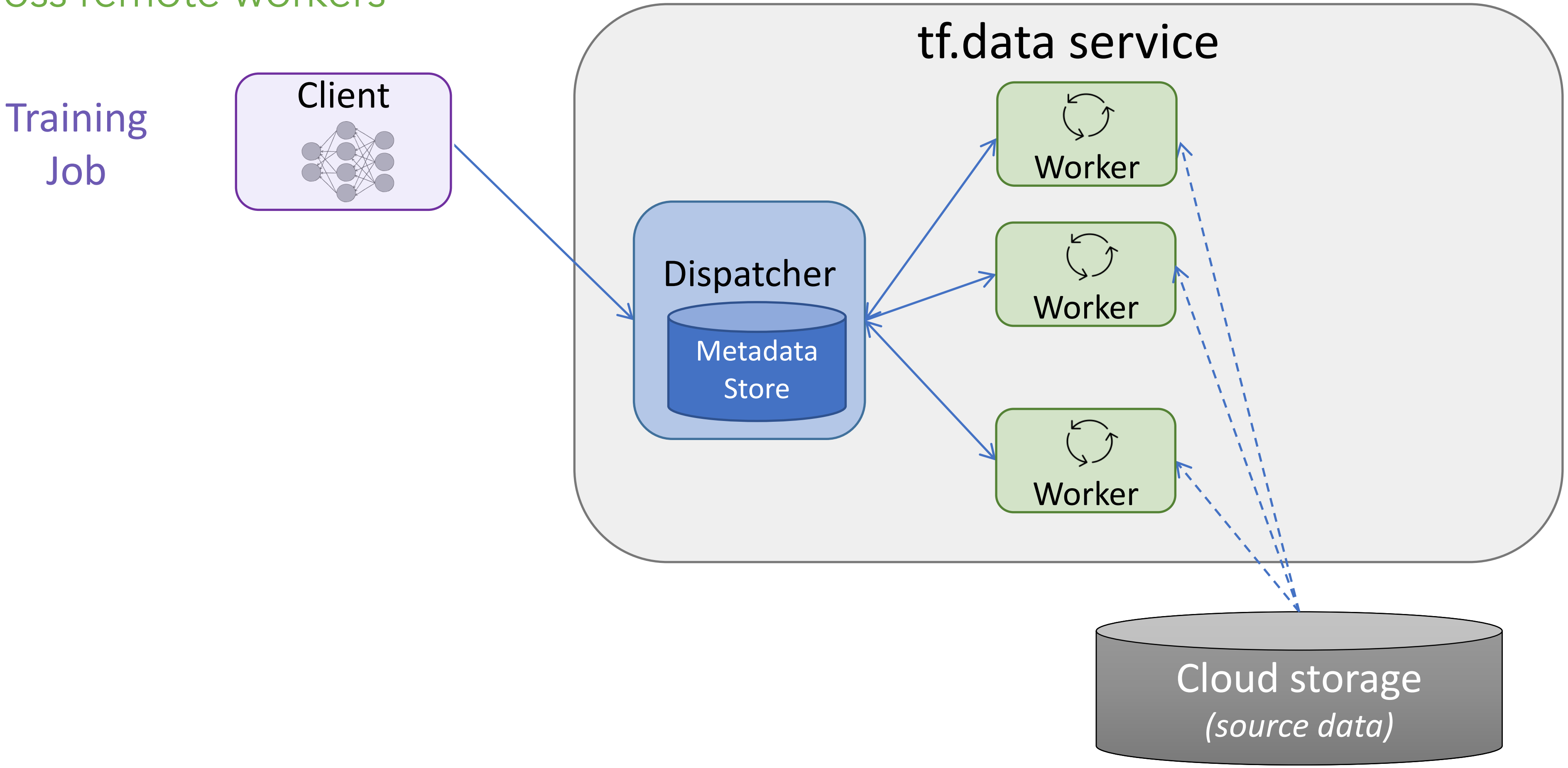
tf.data service: disagg ML data processing

Users register ML data processing job
with the tf.data service dispatcher



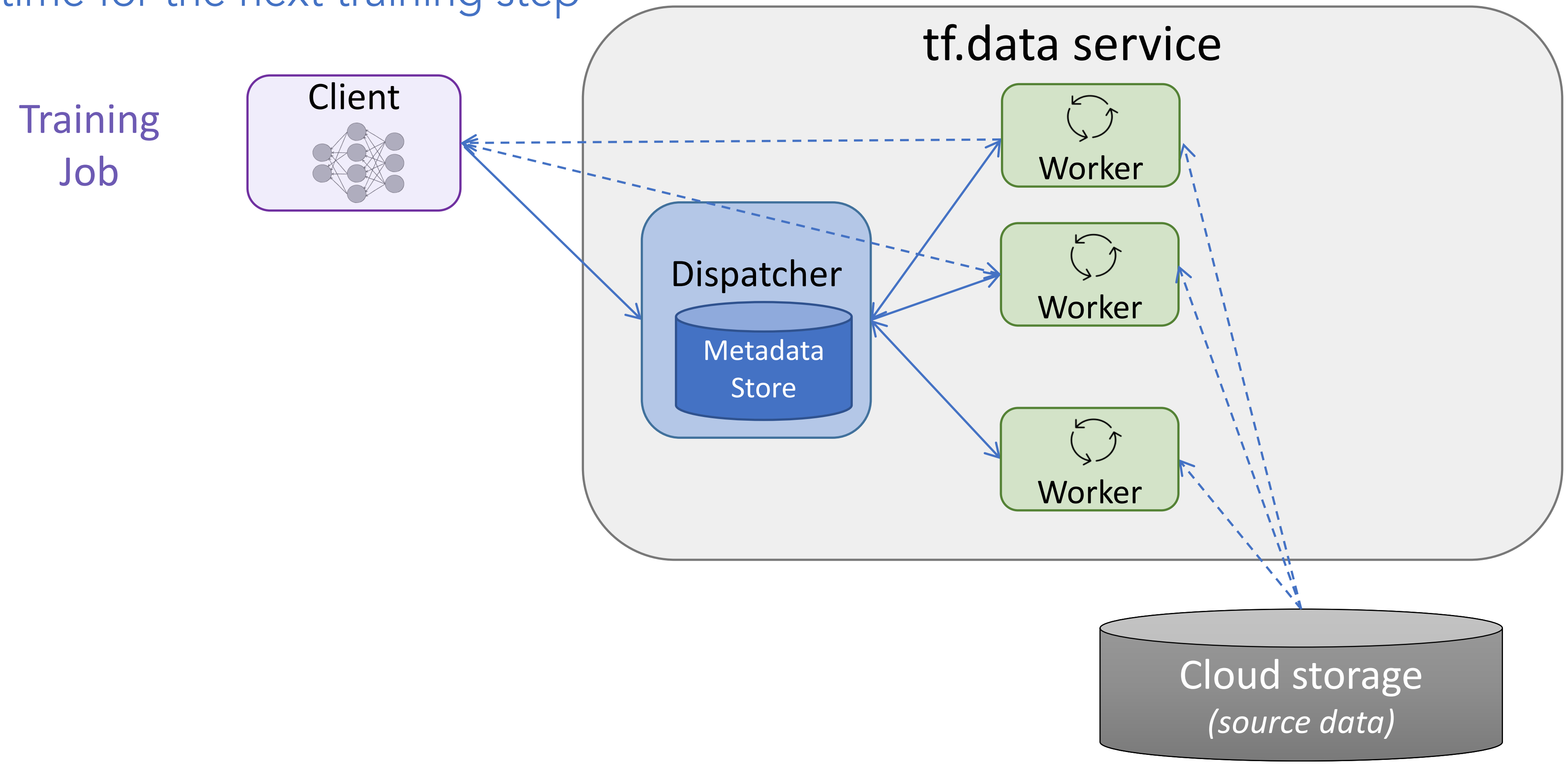
tf.data service: disagg ML data processing

The dispatcher distributes data processing across remote workers



tf.data service: disagg ML data processing

Clients fetch processed data from workers
in time for the next training step



```python
import tensorflow as tf

def preprocess(record):
  ...


dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch()


model = ...
model.fit(dataset, epochs=10)
```

```python
import tensorflow as tf

def preprocess(record):
  ...


dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch()
dataset = dataset.distribute(dispatcher_IP)

model = ...
model.fit(dataset, epochs=10)
```

register input pipeline with dispatcher

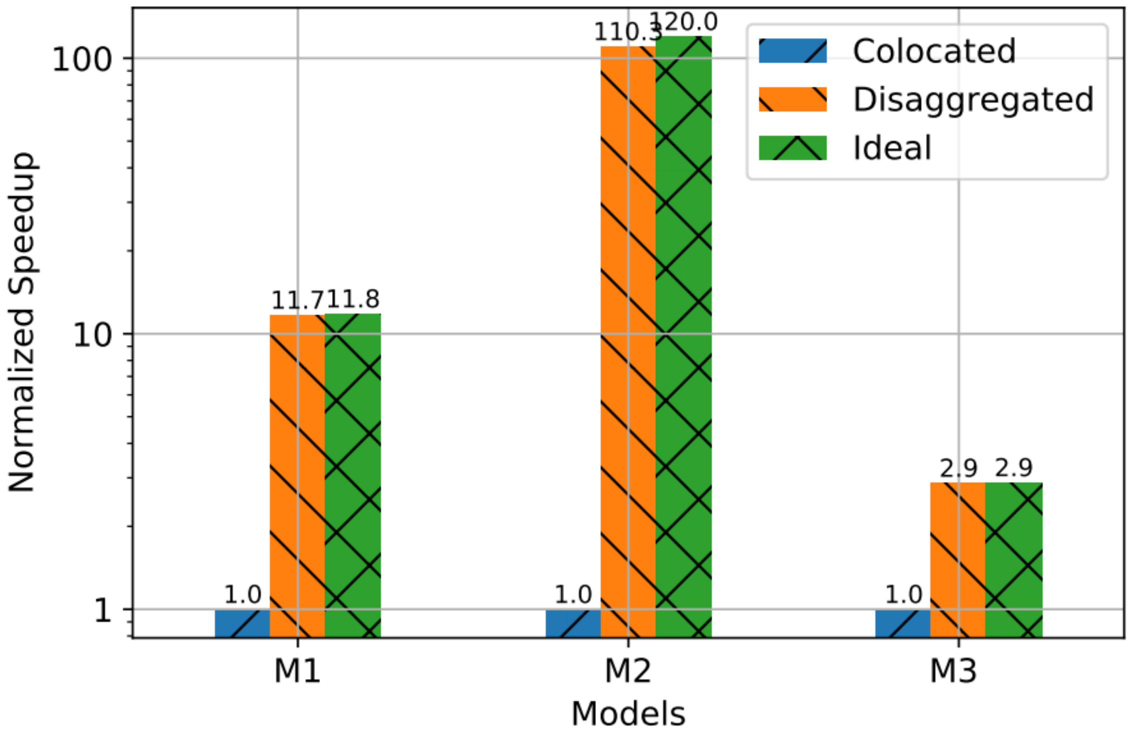
Benefits of disaggregated ML data processing

Remove input bottlenecks

Benefits of disaggregated ML data processing

Remove input bottlenecks → up to **110x** speedup

Training time speedup

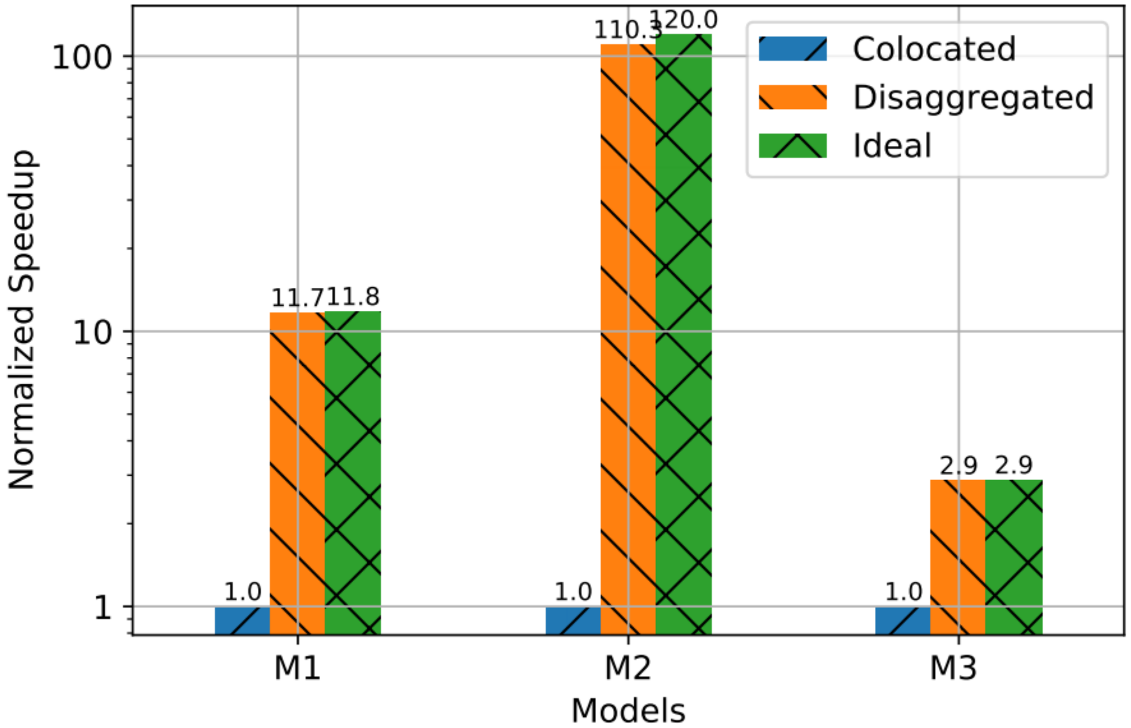


Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiri Simsa, Chandu Thekkath. *A case for disaggregation of ML data processing*, 2022.

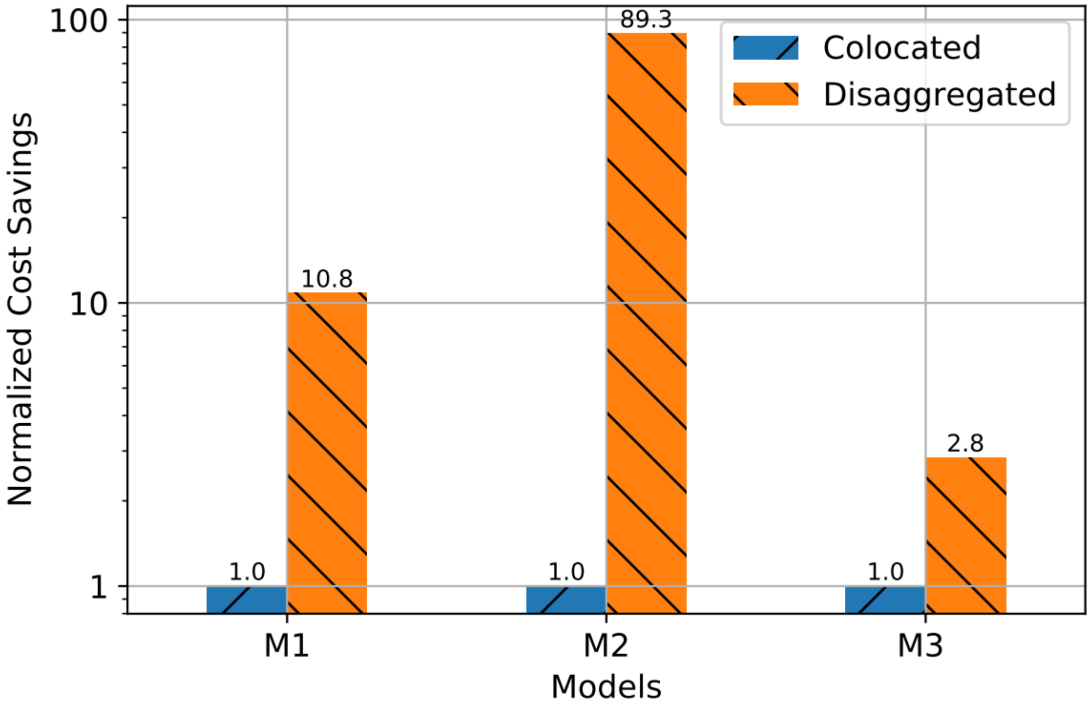
Benefits of disaggregated ML data processing

Remove input bottlenecks → up to **110x** speedup, **89x** cost reduction

Training time speedup



Cost reduction

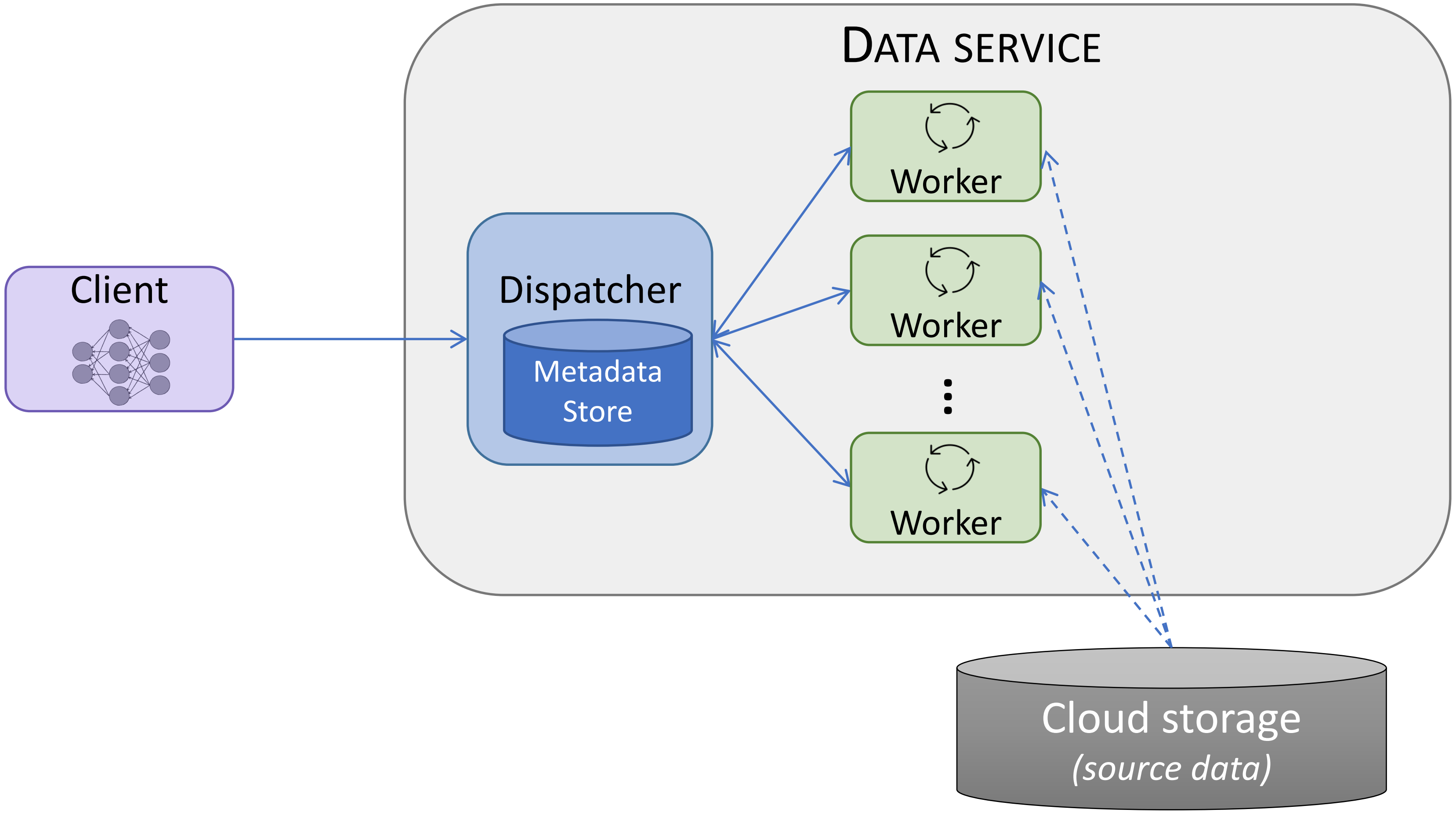


Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiri Simsa, Chandu Thekkath. *A case for disaggregation of ML data processing*, 2022.

How to optimize ML input data processing?

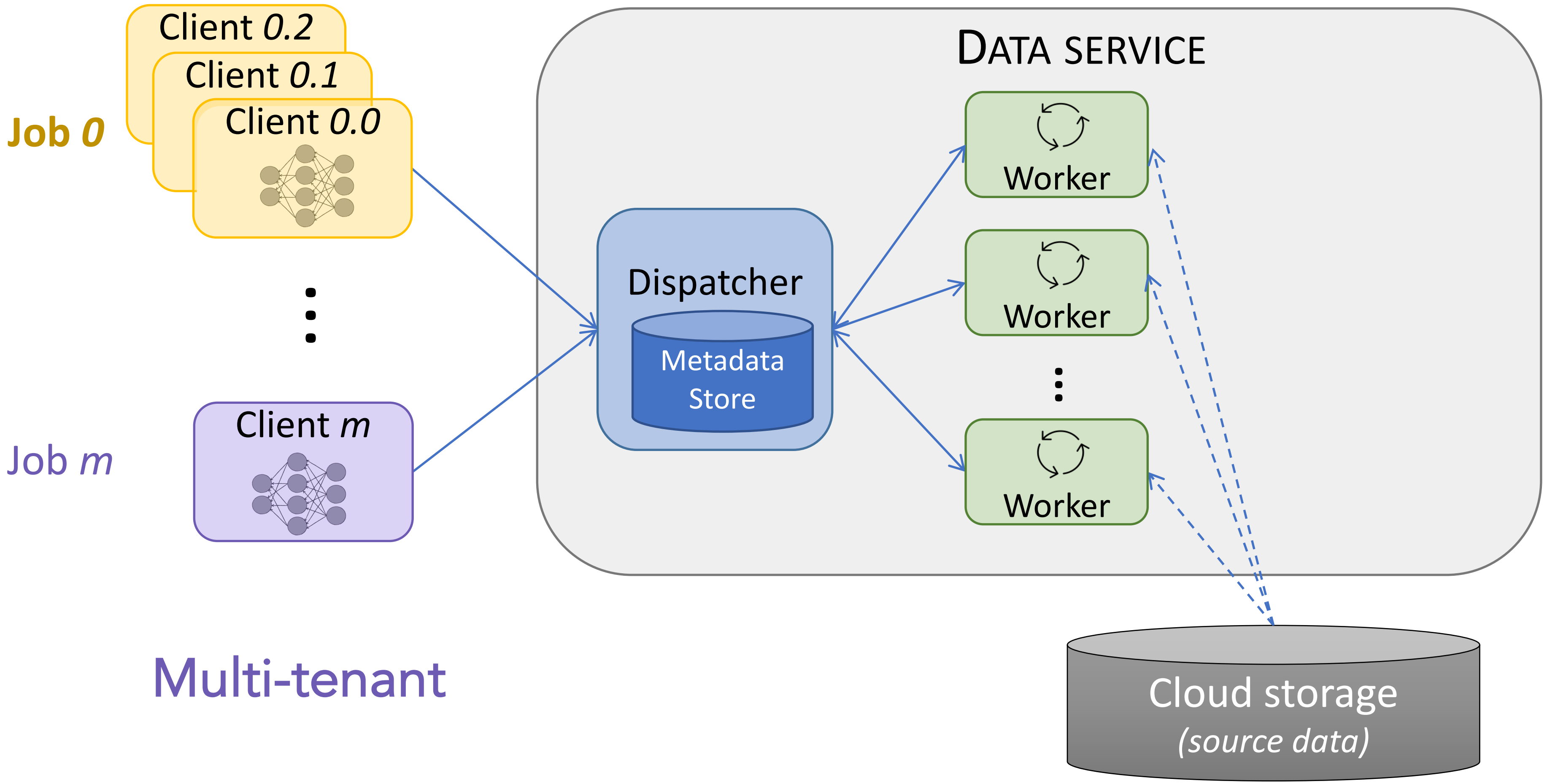
1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service

ML data processing as a service



ML data processing as a service

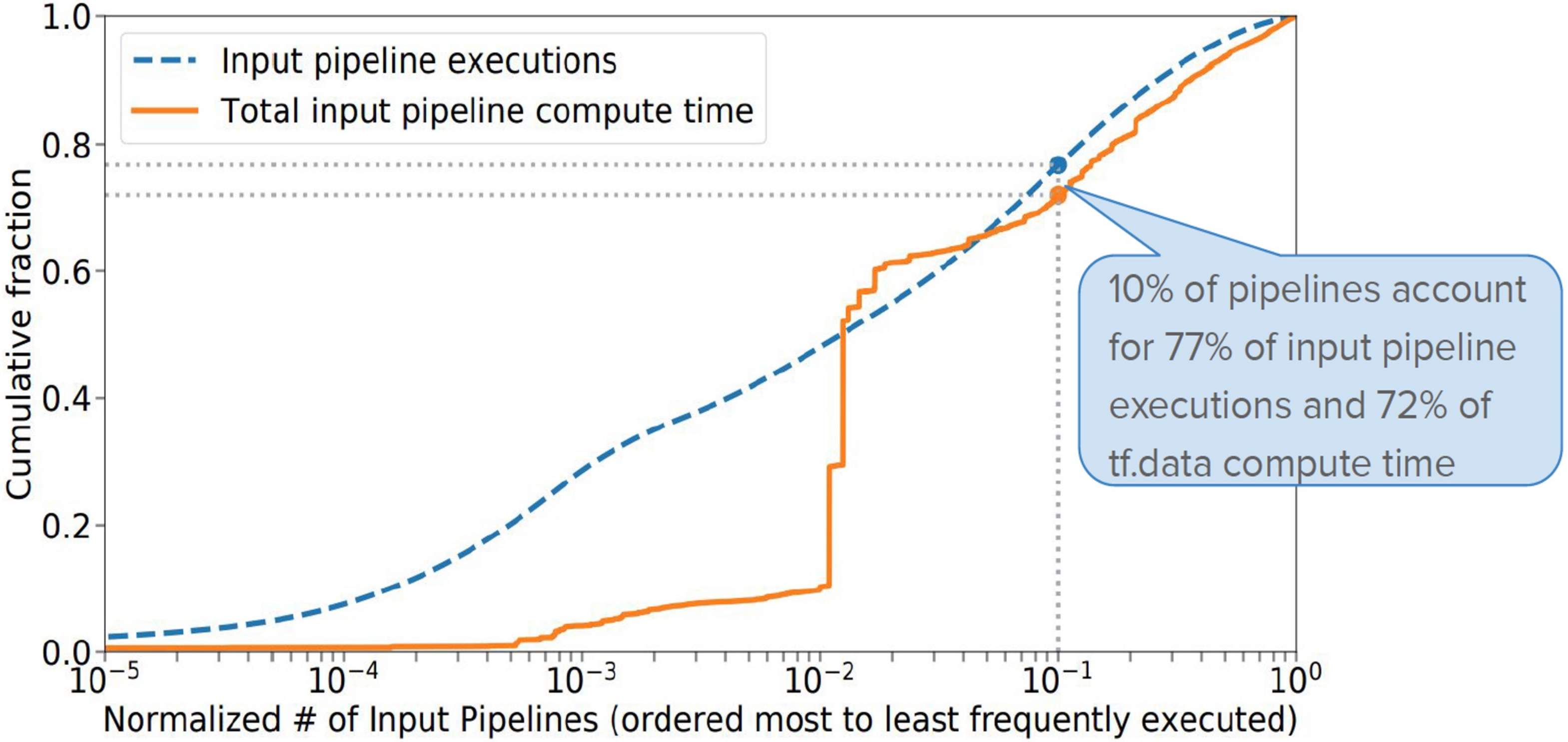
Can we leverage a global view of data processing across jobs?



Multi-tenant

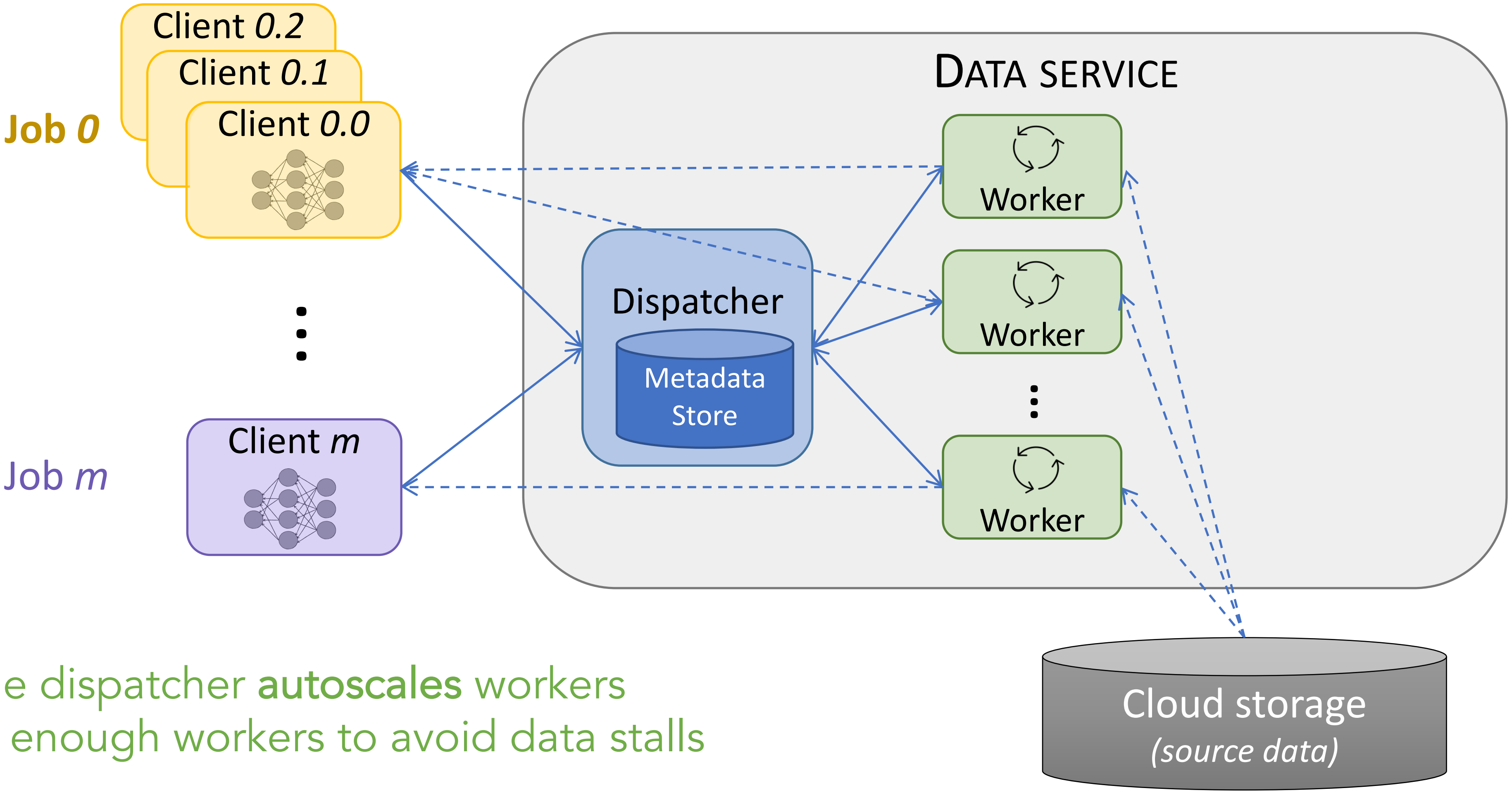
Why leverage knowledge across jobs?

- Input data pipeline are often re-executed across jobs
  - e.g., hyperparameter tuning



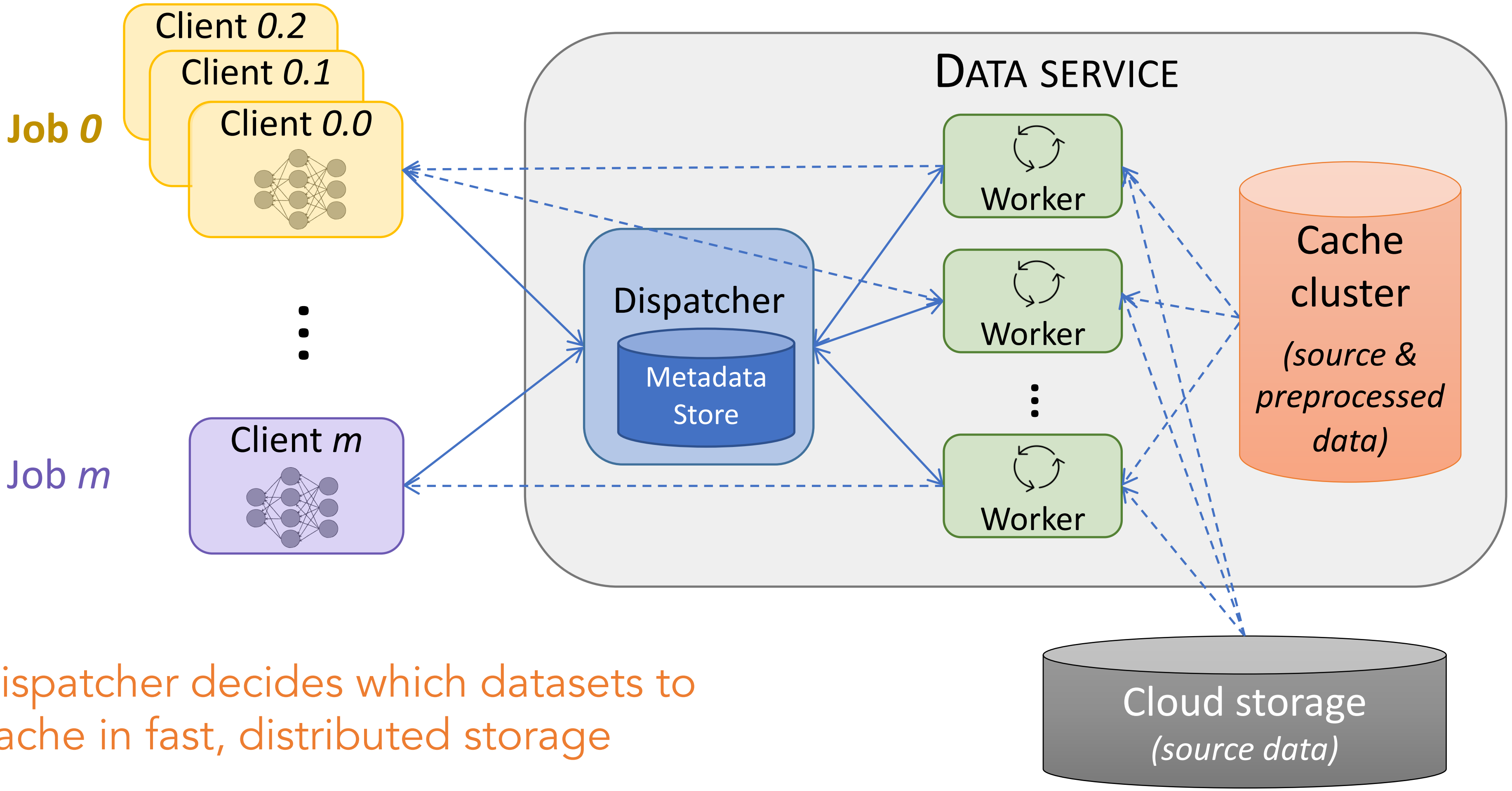
Derek G. Murray, Jiří Šimša, Ana Klimovic, Ihor Indyk: "tf.data: A Machine Learning Data Processing Framework". VLDB 2021.

Cachew: ML data processing as a service



The dispatcher **autoscales** workers
→ just enough workers to avoid data stalls

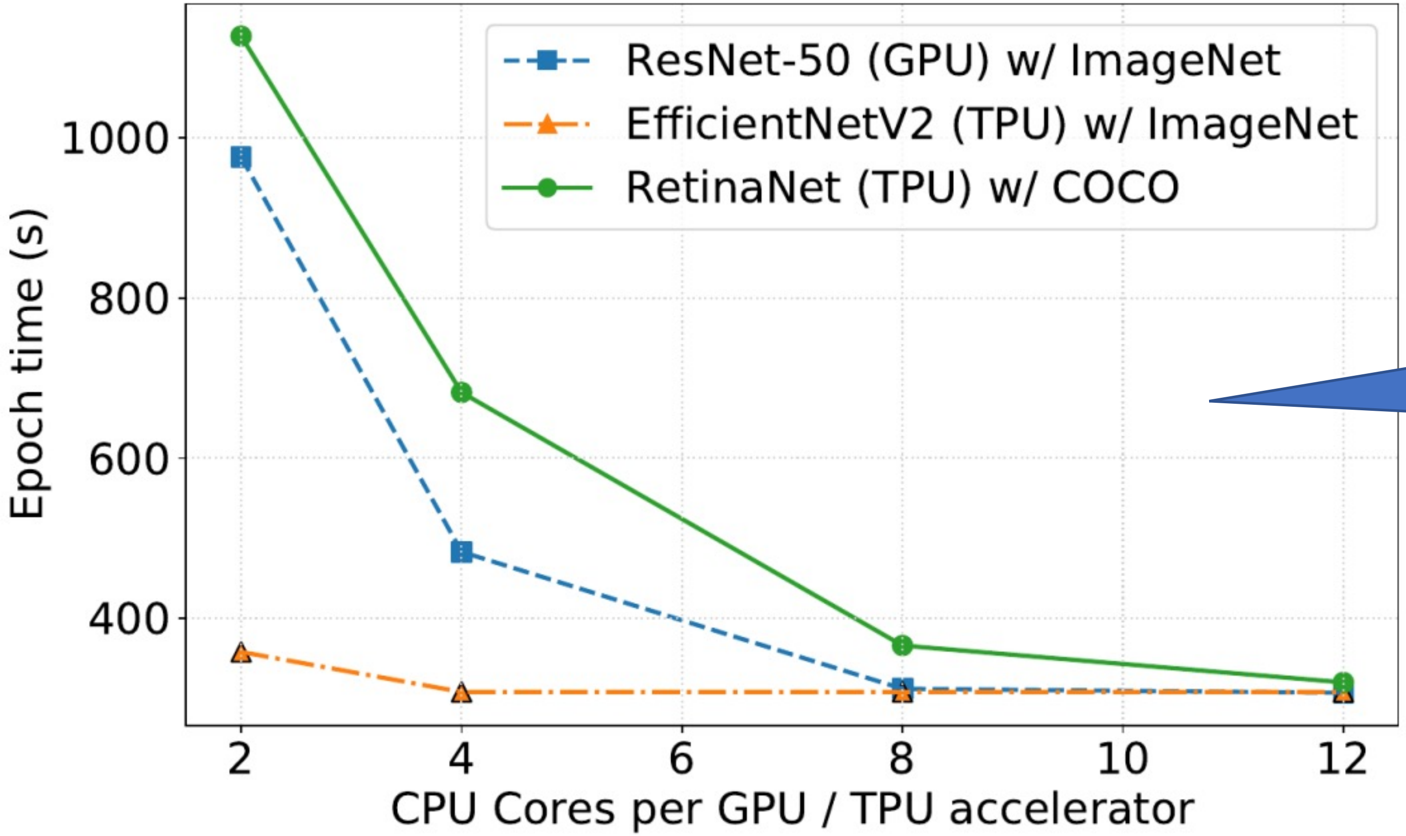
Cachew: ML data processing as a service



The dispatcher decides which datasets to cache in fast, distributed storage

Challenges for ML data processing service

1. How to efficiently autoscale resources for input data processing?



Training jobs benefit differently when given more CPU for data processing per accelerator core.

Challenges for ML data processing service

1. How to efficiently autoscale resources for input data processing?
2. How/when to efficiently cache and re-use (transformed) datasets?

Challenges for ML data processing service

1. How to efficiently autoscale resources for input data processing?
2. How/when to efficiently cache and re-use (transformed) datasets?

Caching does not always improve performance…

- Input data reading may not be the training bottleneck
- Transformed dataset may be much larger than source dataset, saturing cache I/O bandwidth
- Reusing non-deterministically transformed data can hurt ML model accuracy (removes randomness)

Challenges for ML data processing service

1. How to efficiently autoscale resources for input data processing?
2. How/when to efficiently cache and re-use (transformed) datasets?

Scaling & caching are difficult optimization decisions for users.

Opportunity for ML data processing service

1. How to efficiently **autoscale resources** for input data processing?
2. How/when to efficiently **cache and re-use** (transformed) datasets?

Scaling & caching are difficult optimization decisions for users.

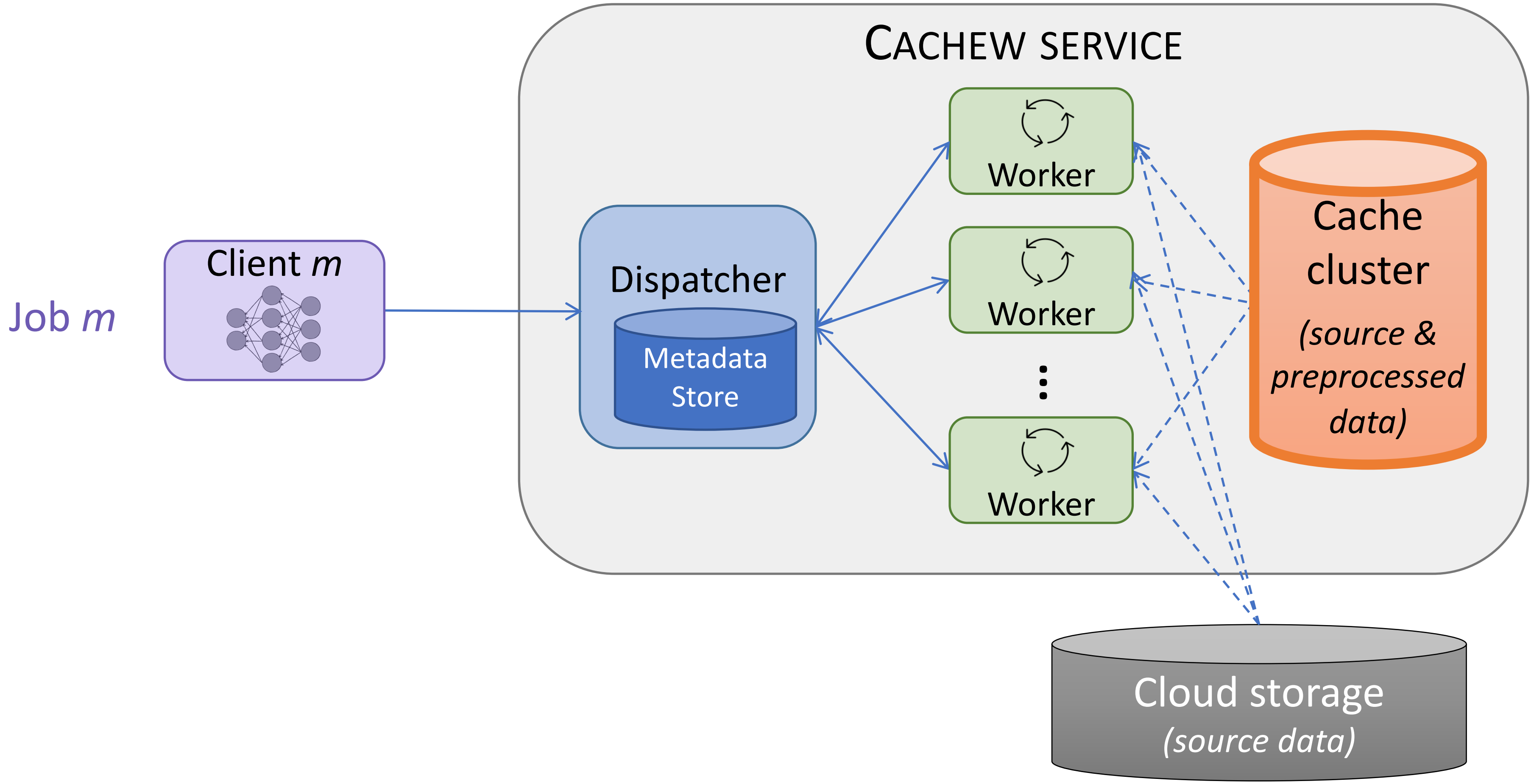
→ Need a data processing service that automates these decisions.

Cachew

https://github.com/eth-easl/cachew

Autocaching policy

How to decide whether to read/write a dataset in faster, more $ storage?



```python
import tensorflow as tf

def preprocess(record):
  ...


dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(parse).filter(filter_func).map(rand_augment)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch()
dataset = dataset.distribute(dispatcher_IP)

model = ...
model.fit(dataset, epochs=10)
```

```python
import tensorflow as tf

def preprocess(record):
  ...

dataset = tf.data.TFRecordDataset("...*.tfrecord")
dataset = dataset.map(parse).filter(filter_func).map(rand_augment)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch()
dataset = dataset.distribute(dispatcher_IP)

model = ...
model.fit(dataset, epochs=10)
```

user-defined preprocessing

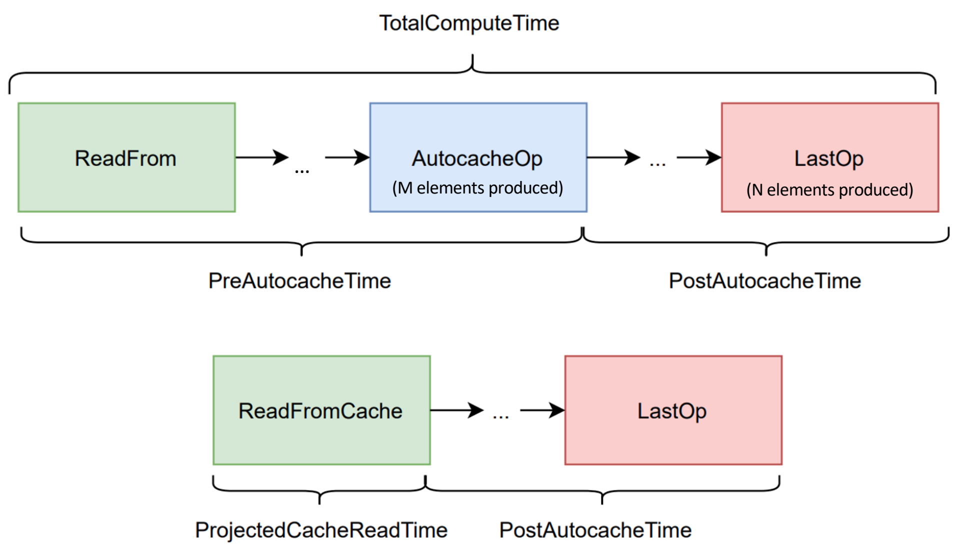
```python
import tensorflow as tf

def preprocess(record):
  ...


dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.autocache().map(parse).filter(filter_func).autocache().map(rand_augment)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch()
dataset = dataset.distribute(dispatcher_IP)

model = ...
model.fit(dataset, epochs=10)
```

Cachew users can apply autocache ops to **hint** where it is viable (from an *ML perspective*) to cache/reuse data

Cachew will decide which autocache op is an optimal dataset to cache from a ***throughput perspective***. Caching will only be applied at 1 location, if at all.

Autocaching policy

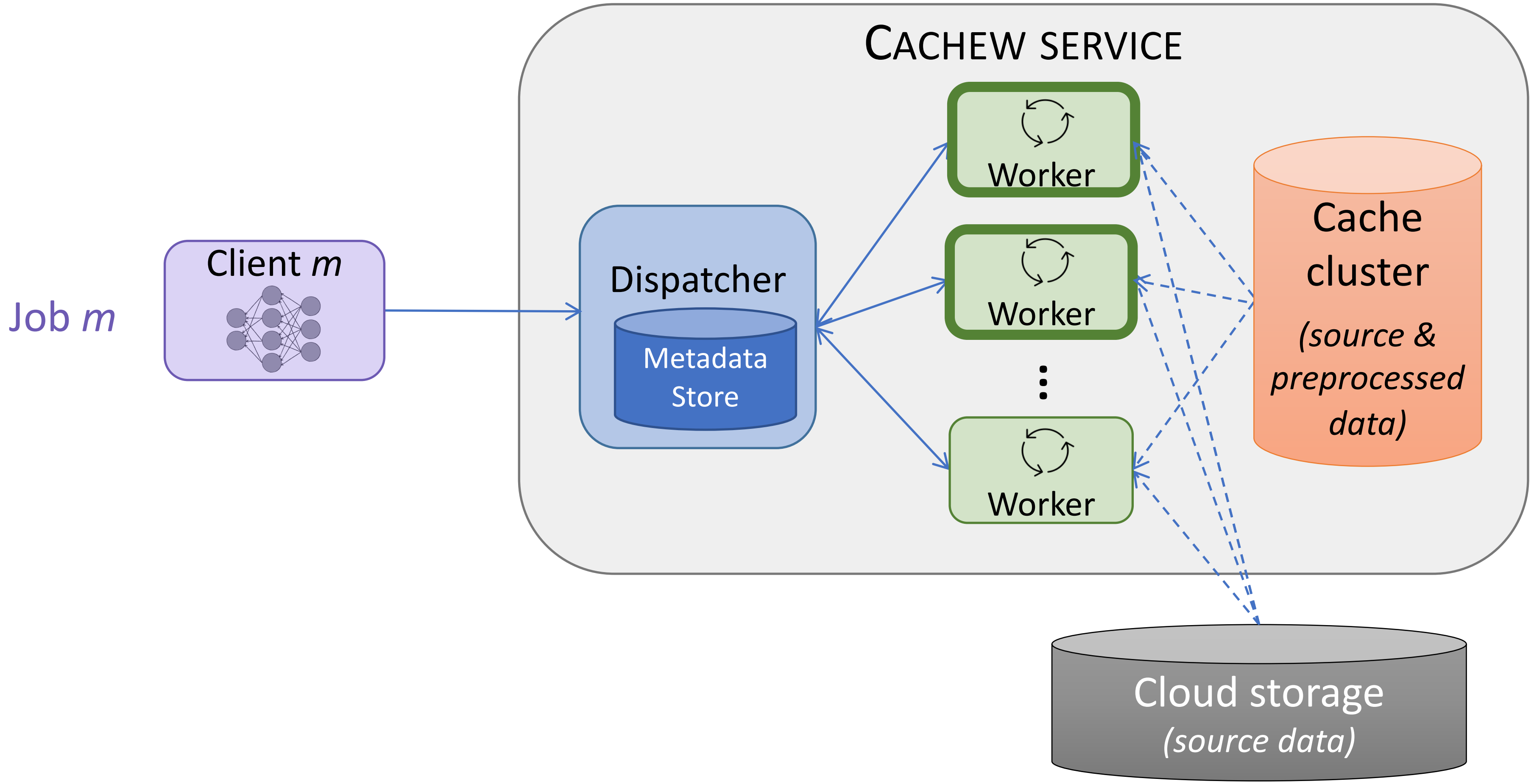
- During first epoch, at each **autocache** op, infer *compute* vs. *cache* read throughput:



- Cachew selects the **autocache** op with max throughput (i.e. min *TotalCacheExecTime*)
- Compare with the throughput of pure compute (*TotalComputeTime*)
- Select option with highest throughput → at most one **autocache** selected

Autoscaling policy

How to decide how many workers to allocate for a job?



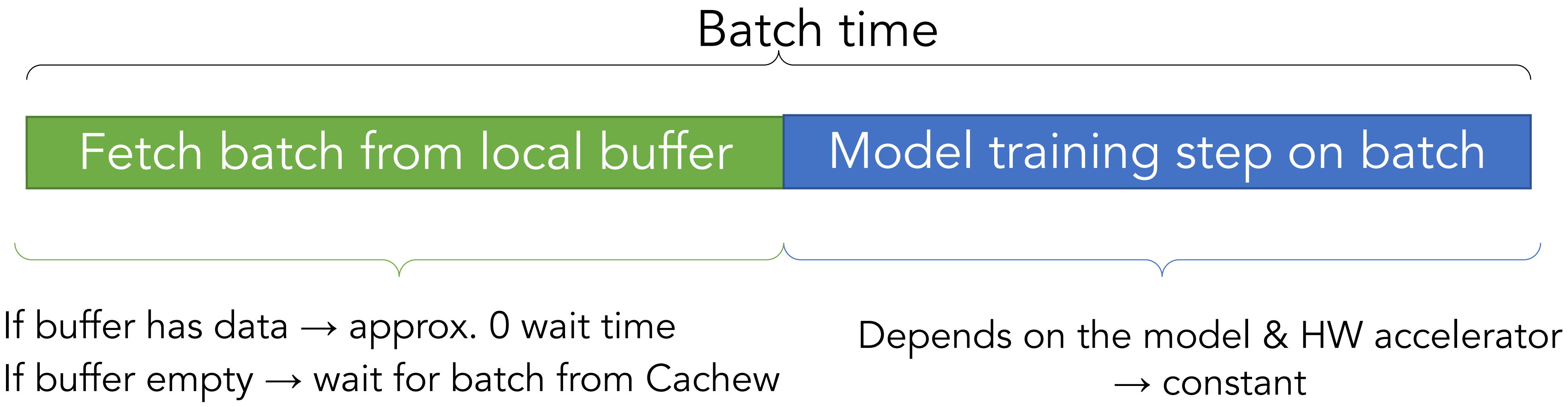
Autoscaling policy

- Intuition: scale up data workers until no additional benefit to end-to-end training time.

- How to estimate end-to-end training time as we scale workers?
  - Leverage the iterative nature of ML training: monitor batch time

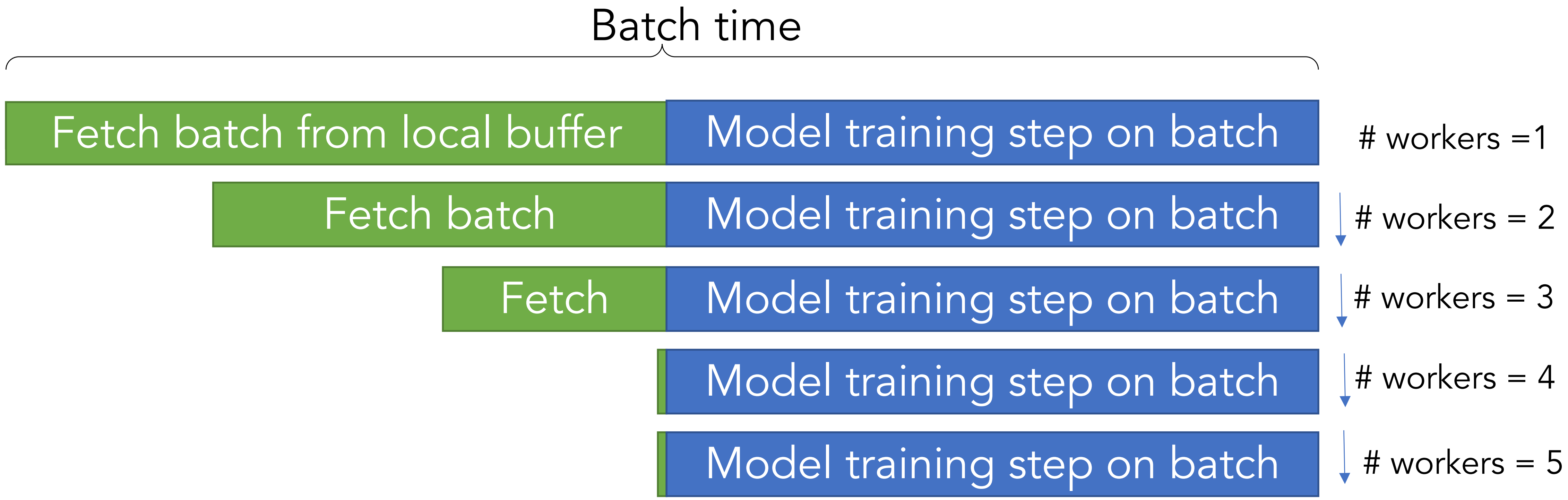
Autoscaling policy

- Intuition: scale up data workers until no additional benefit to end-to-end training time.



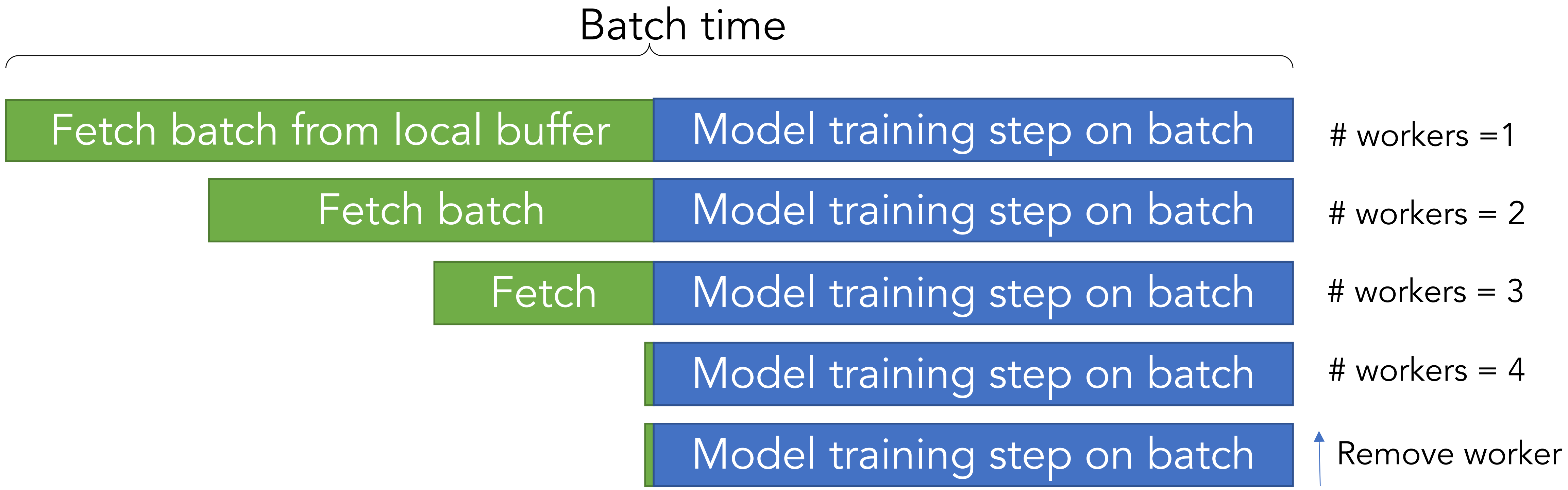
Autoscaling policy

- Intuition: scale up data workers until no additional benefit to end-to-end training time.



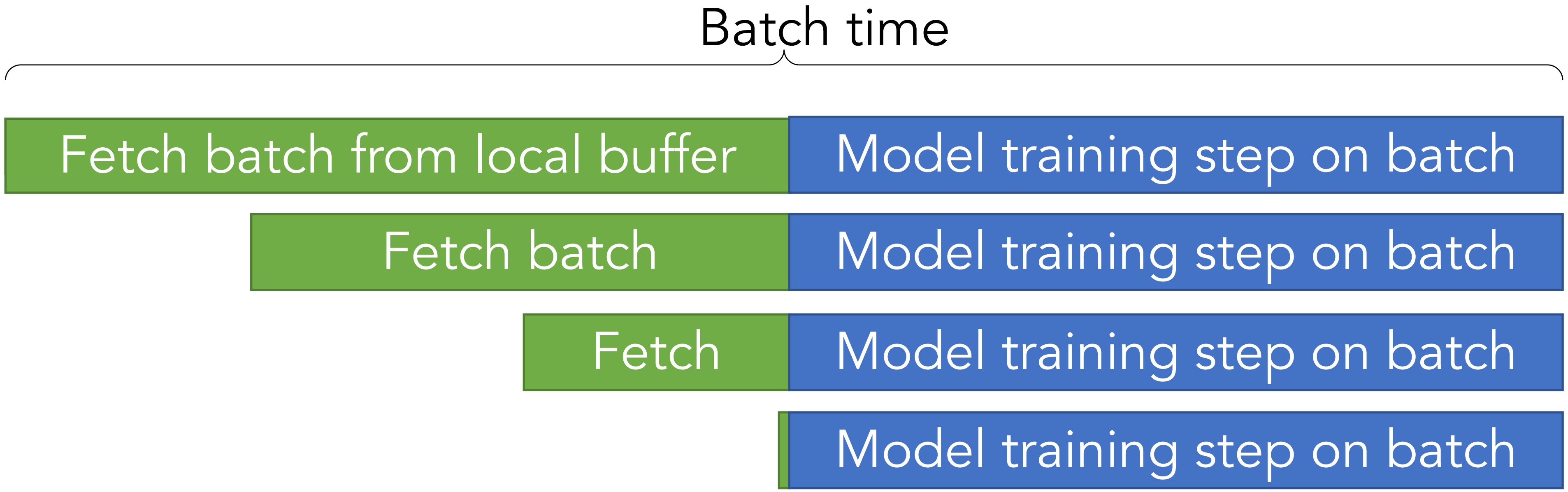
Autoscaling policy

- Intuition: scale up data workers until no additional benefit to end-to-end training time.



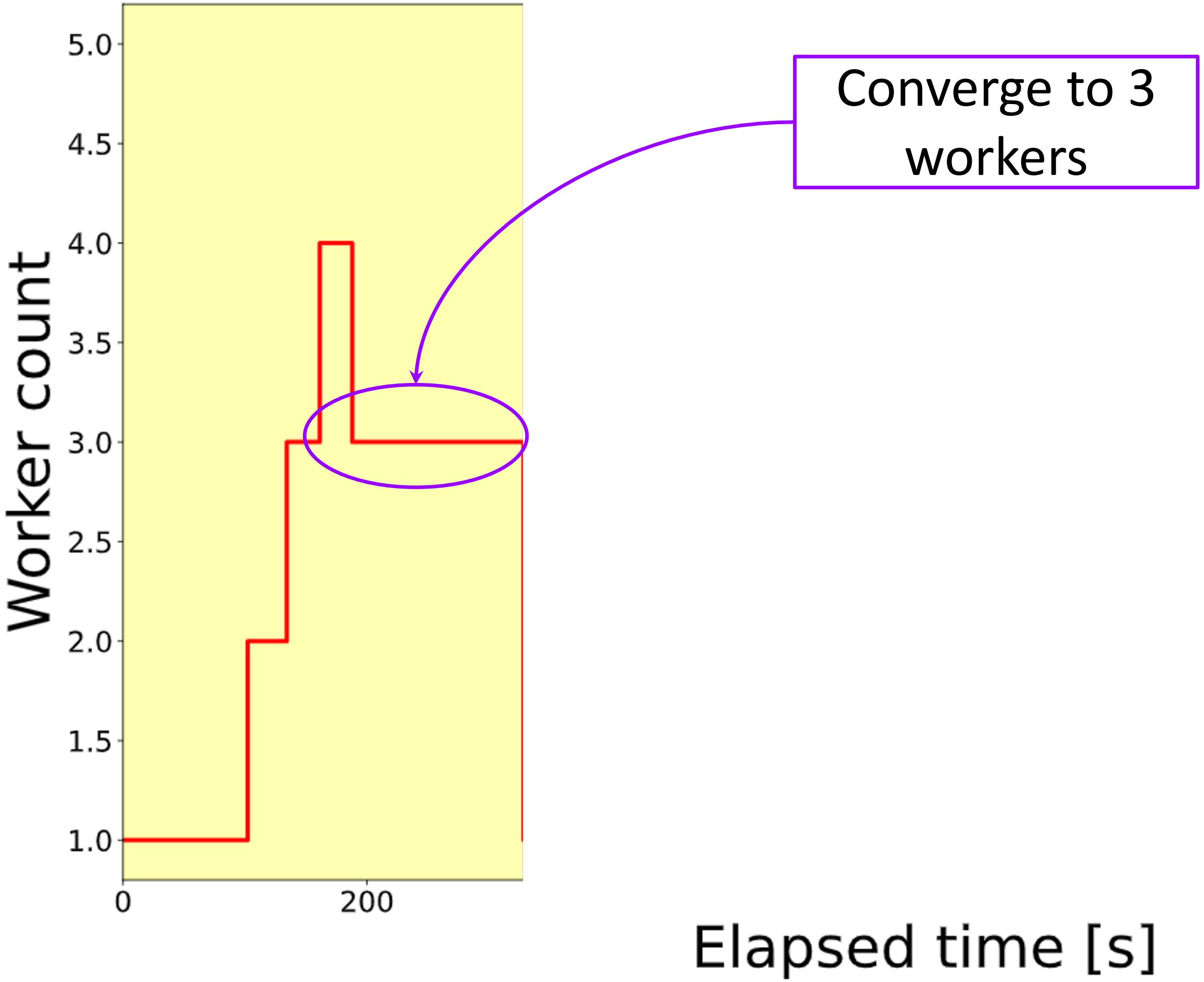
Autoscaling policy

- Intuition: scale up data workers until no additional benefit to end-to-end training time.

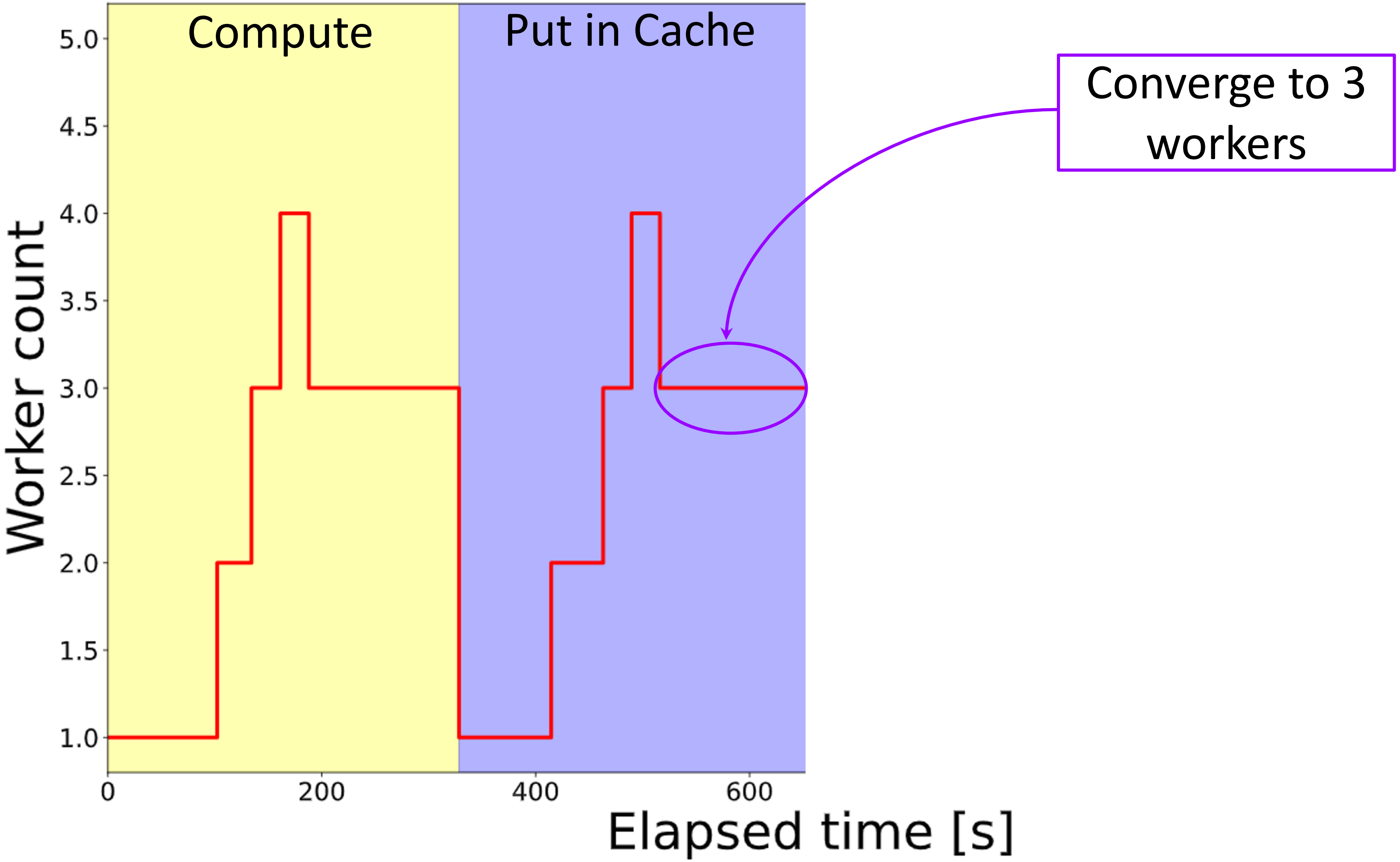


Converged:
workers = 4

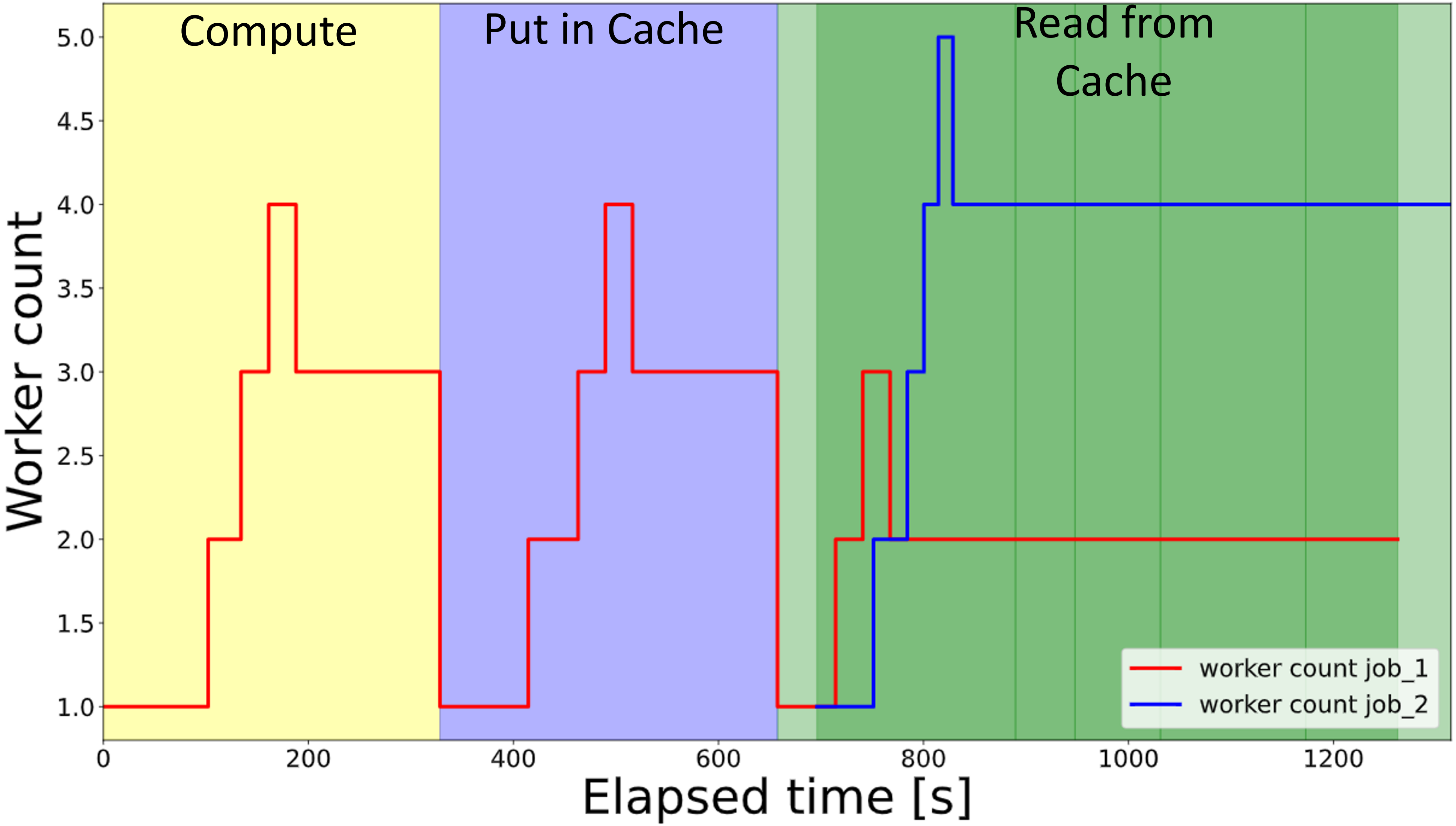
Cachew autoscaling & caching for multiple tenants



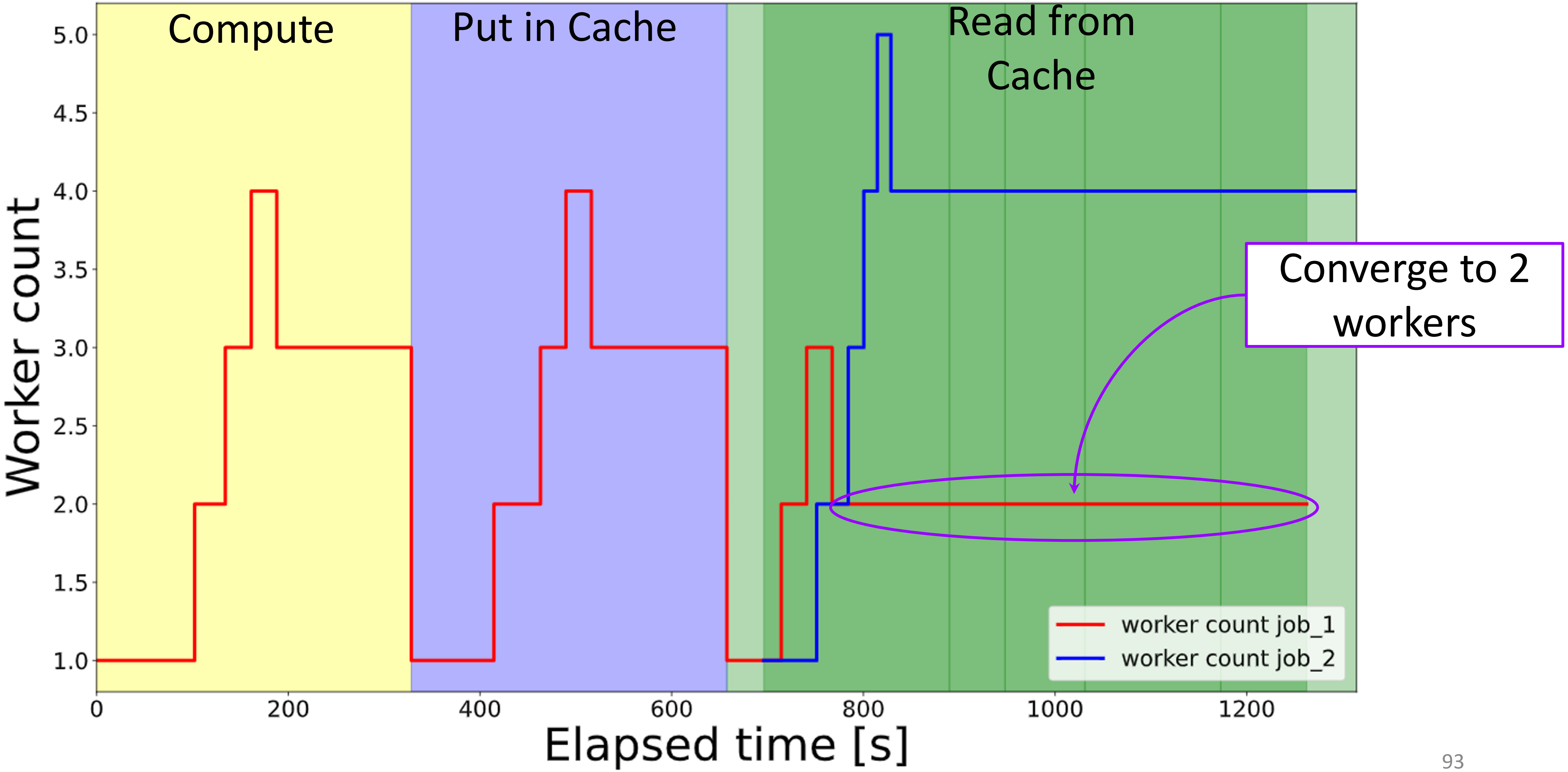
Cachew autoscaling & caching for multiple tenants



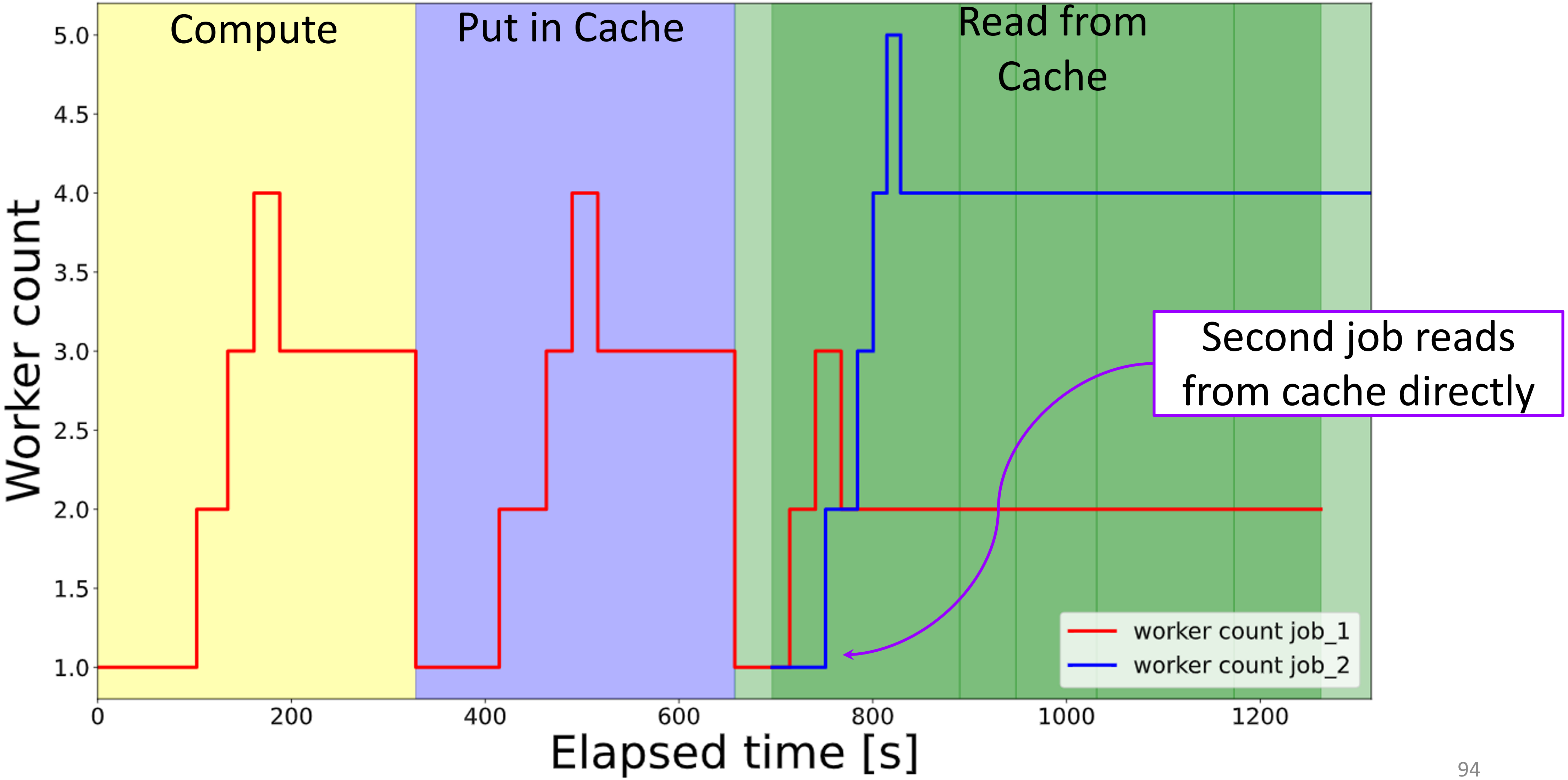
Cachew autoscaling & caching for multiple tenants



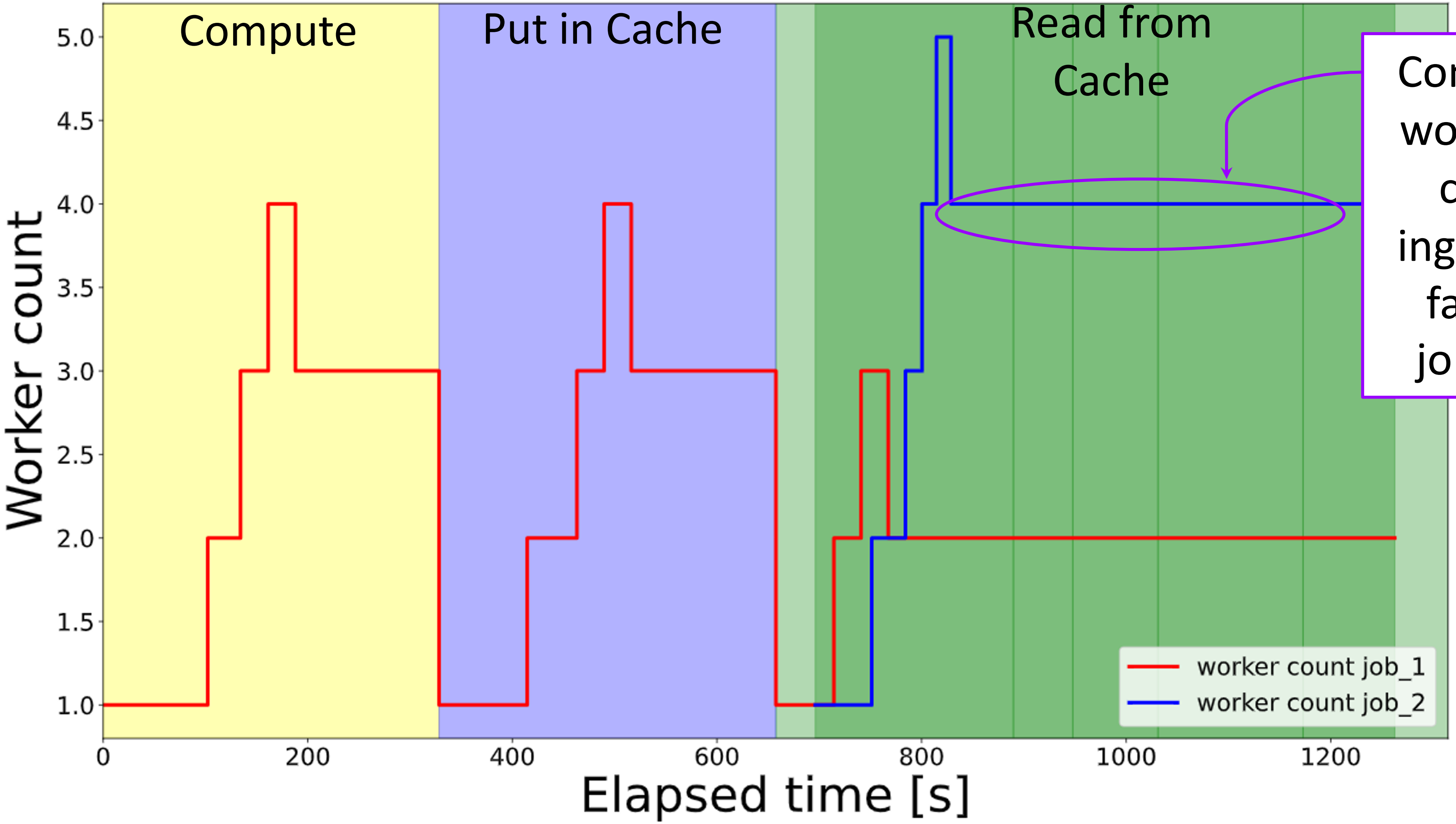
Cachew autoscaling & caching for multiple tenants



Cachew autoscaling & caching for multiple tenants



Cachew autoscaling & caching for multiple tenants



Converge to 4 workers; job2 client can ingest data 2x faster than job1's client

Future directions for ML data services

How to leverage knowledge across jobs to improve data and model *quality*?

- Training data discovery service
  - Recommend “relevant” source datasets used by other jobs
- Data auto-augmentation service
  - Recommend data augmentations
- Data importance service
  - Recommend training examples that are most relevant for the task at hand

ML with *dynamic* input datasets

- Many practical ML use-cases involve training on dynamic data:
  - New data streaming in, some older data needs to be deleted

  → Model needs to adapt; learn from new data + recall "important" old data

- Need system support for:
  - Efficiently mixing new (streaming) & old (stored) data
  - Data importance aware data storage/caching & training
  - Data drift aware model retraining and deployment strategies

- To stimulate research in this area, we are building a open-source benchmark and system architecture for ML training on dynamic datasets.

  → early stage, collaborators welcome!

Google

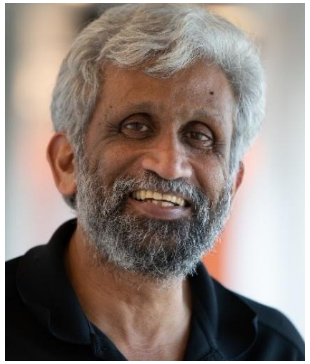
Thanks to great collaborators ☺

ETH zürich

Carnegie Mellon University


Dan Graur


Damien Aymon


Chandu Thekkath


Jiří Šimša


Derek Murray


Dan Kluser


Andrew Audibert


Michael Kuchnik


George Amvrosiadis

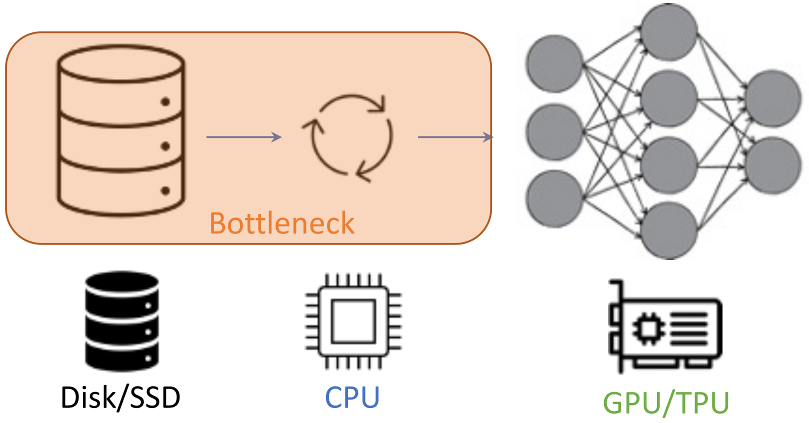

Virginia Smith

ML has a **cost problem**

…to train a 100 Trillion parameter model for 1 day on the cloud?

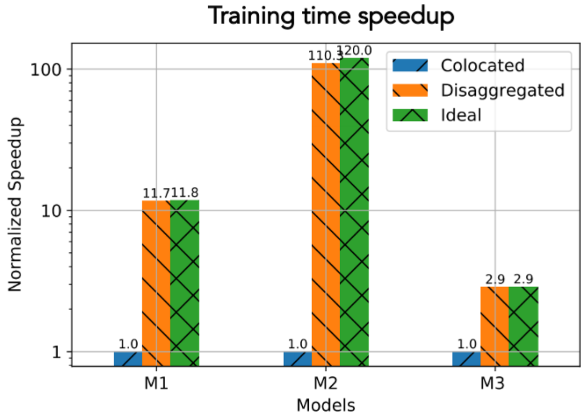
A. $4,000
B. $40,000
C. $400,000

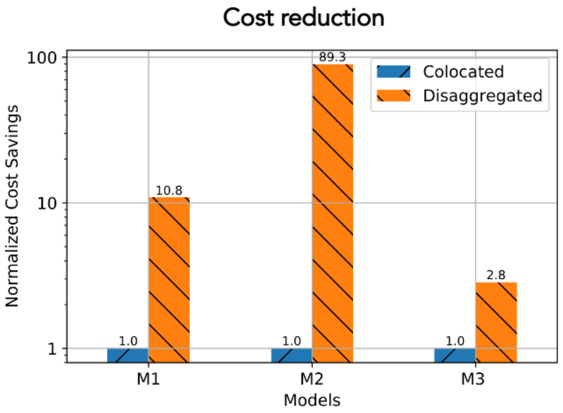
Input data processing is often a **bottleneck,** leaving expensive GPUs/TPUs idle



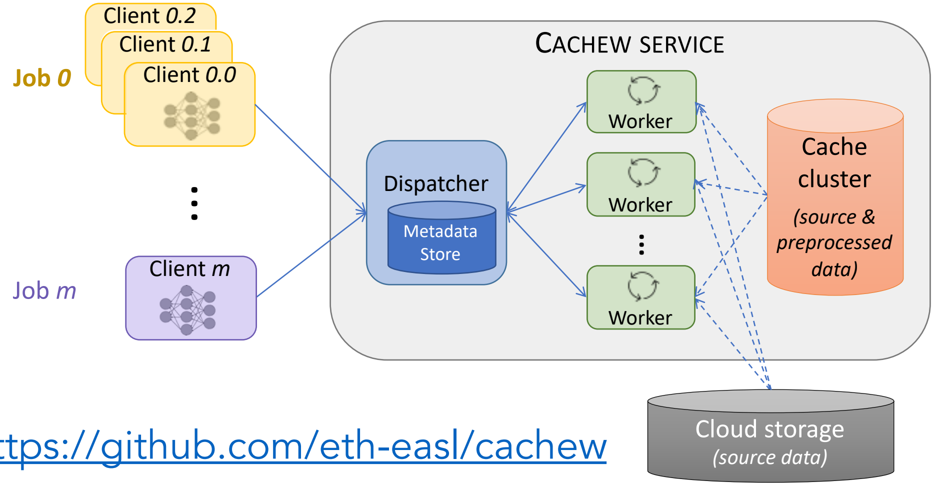
Disaggregating data processing can eliminate data stalls

→ Up to 110x speedup, 89x cost reduction on production model





Cachew: multi-tenant ML data processing service → **autoscale** & **autocache**



https://github.com/eth-easl/cachew