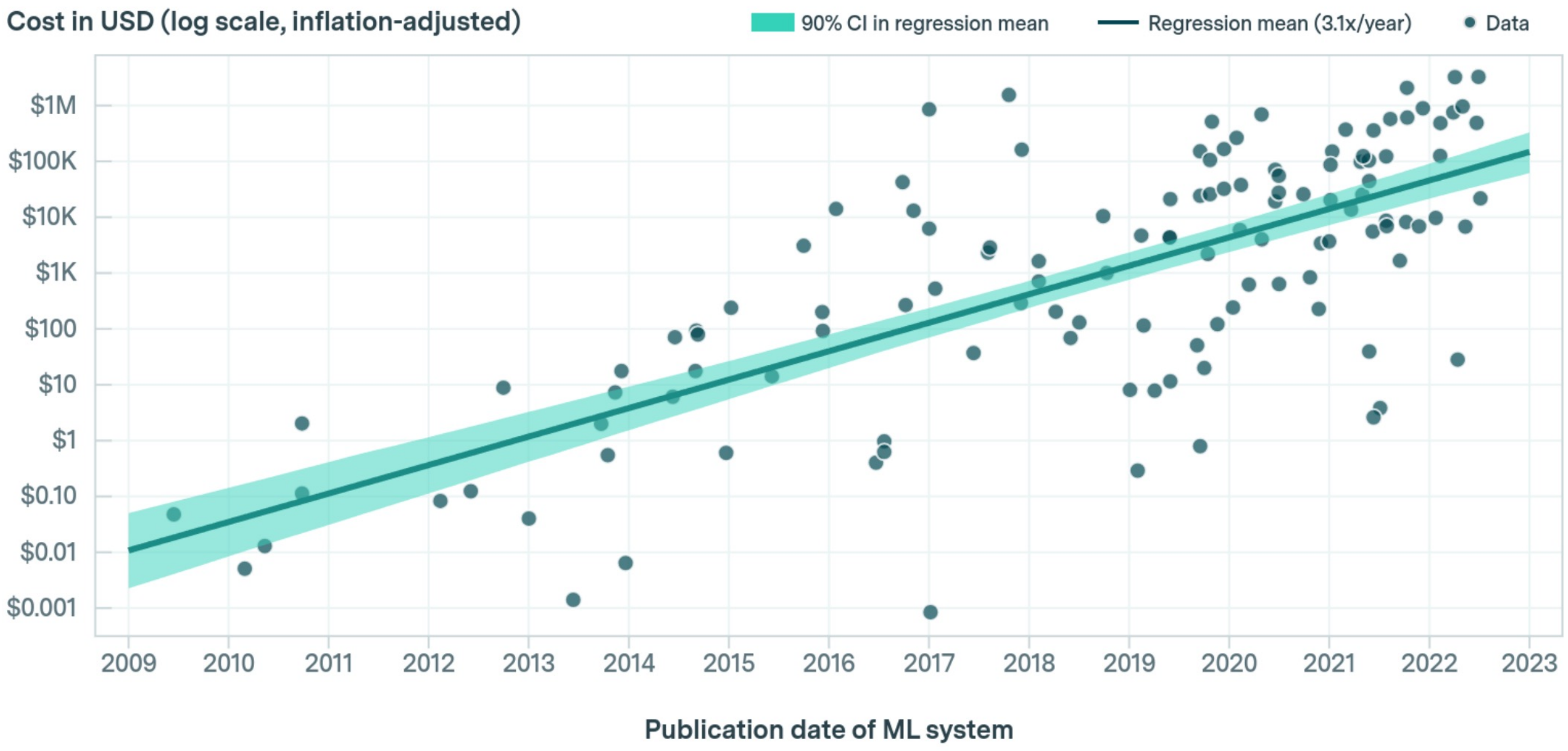
Data Management for Cost-Efficient ML

Ana Klimovic

ETH zürich

EuroSys CHEOPS
April 2024

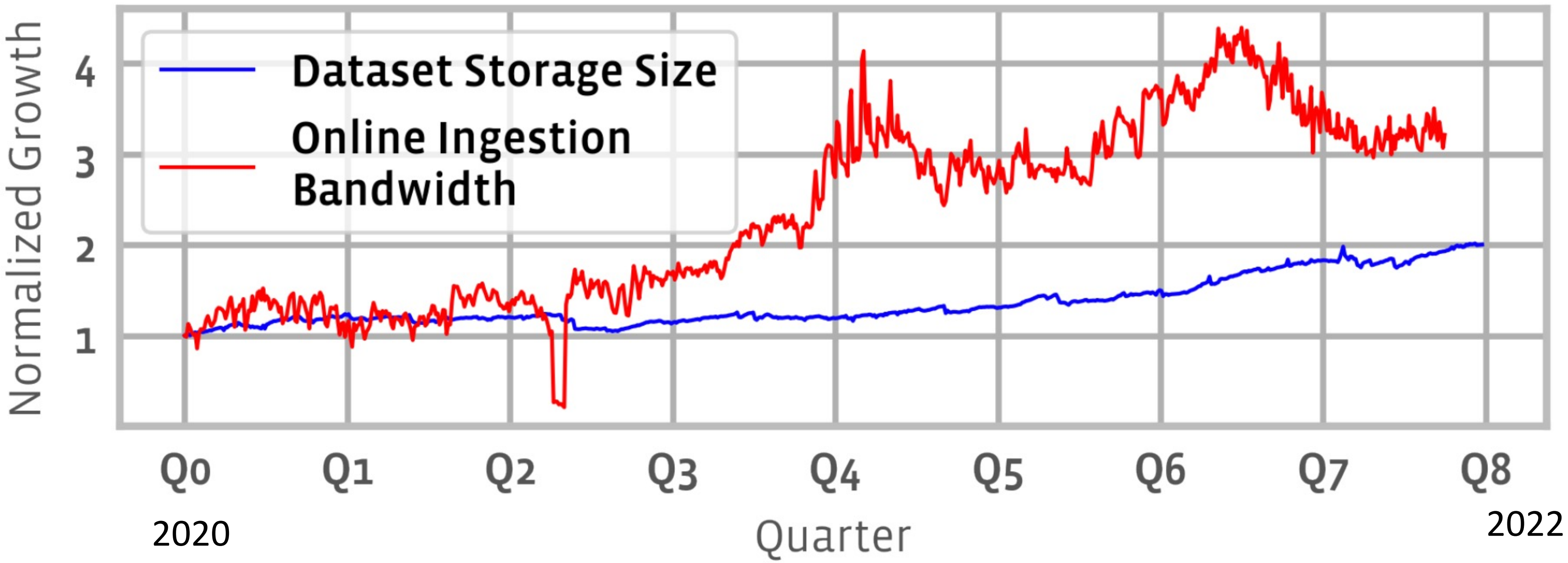
ML training is increasingly expensive



Source: Ben Cottier, 2023. https://epochai.org/blog/trends-in-the-dollar-training-cost-of-machine-learning-systems

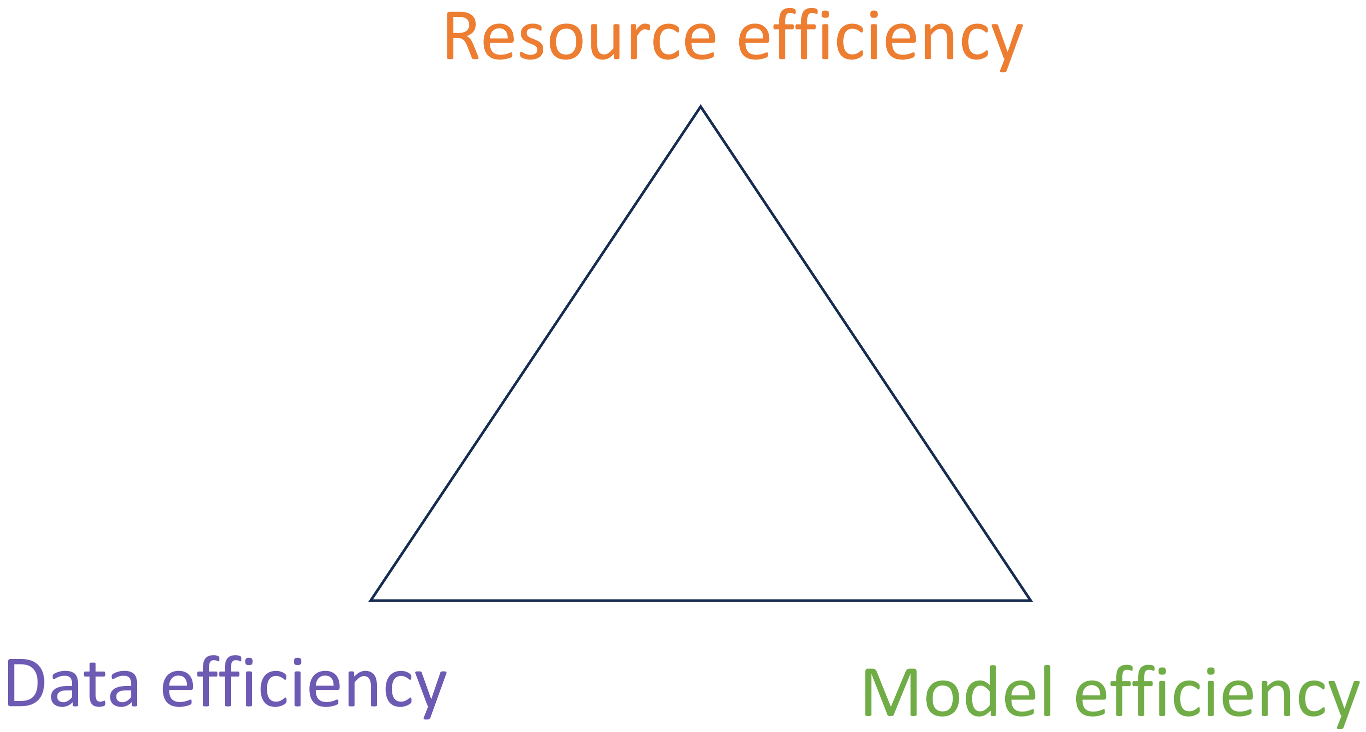
ML training is increasingly data hungry

At Meta, ML data storage and data ingestion bandwidth grew over 2x and 4x in 2 years.



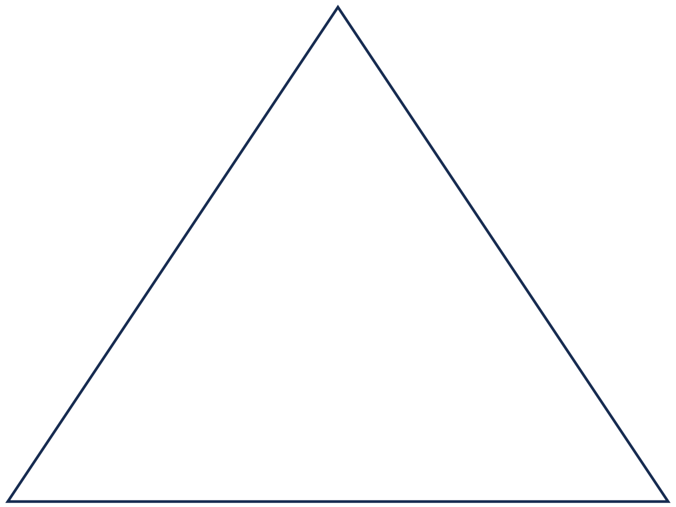
Mark Zhao et al. "**Understanding data storage and ingestion for large-scale deep recommendation model training**", ISCA 2022.

How can we reduce the cost of ML?



How can we reduce the cost of ML?

Resource efficiency → maximize GPU/TPU utilization



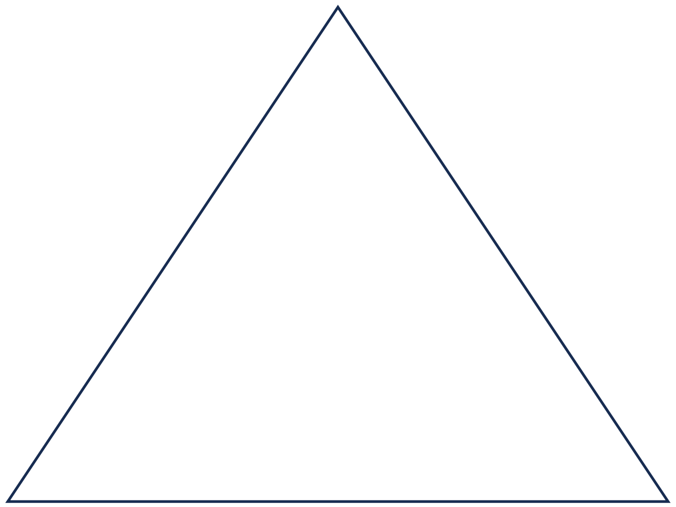
Data efficiency

→ train on the most important data

Model efficiency

How can we reduce the cost of ML?

Resource efficiency → maximize GPU/TPU utilization



Need to optimize how we store & ingest data!

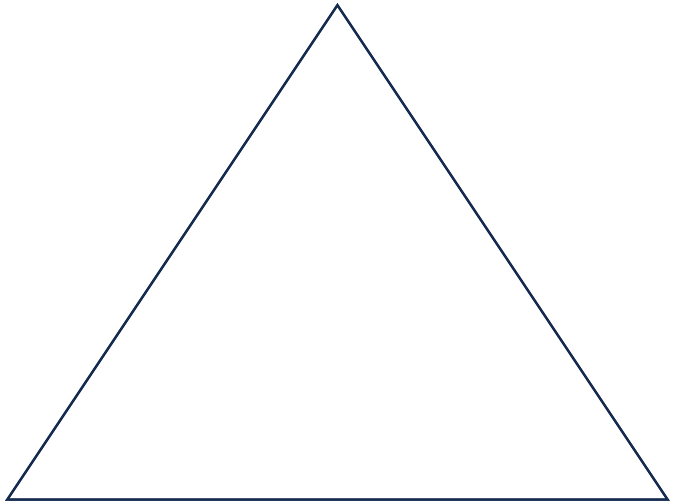
Data efficiency

→ train on the most important data

Model efficiency

How can we reduce the cost of ML?

Resource efficiency → maximize GPU/TPU utilization



Data efficiency

→ train on the most important data

Model efficiency

Need to optimize how we store & ingest data!

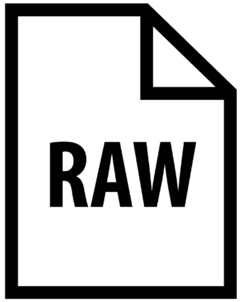
What hinders high GPU/TPU utilization?

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

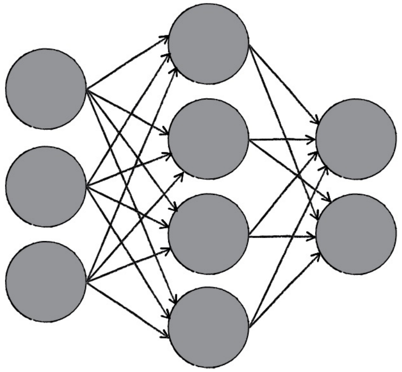
Input data processing for ML

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

Raw Data

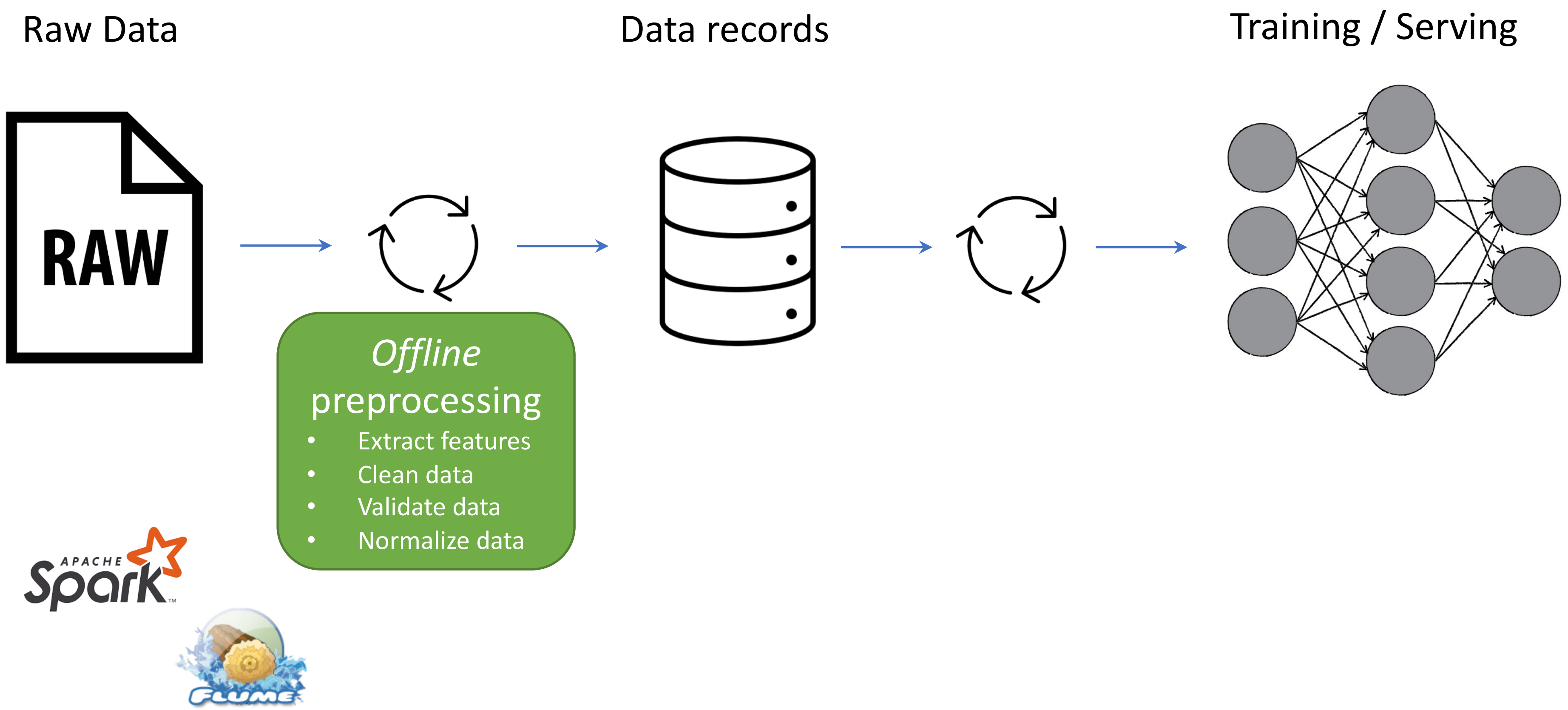


Training / Serving



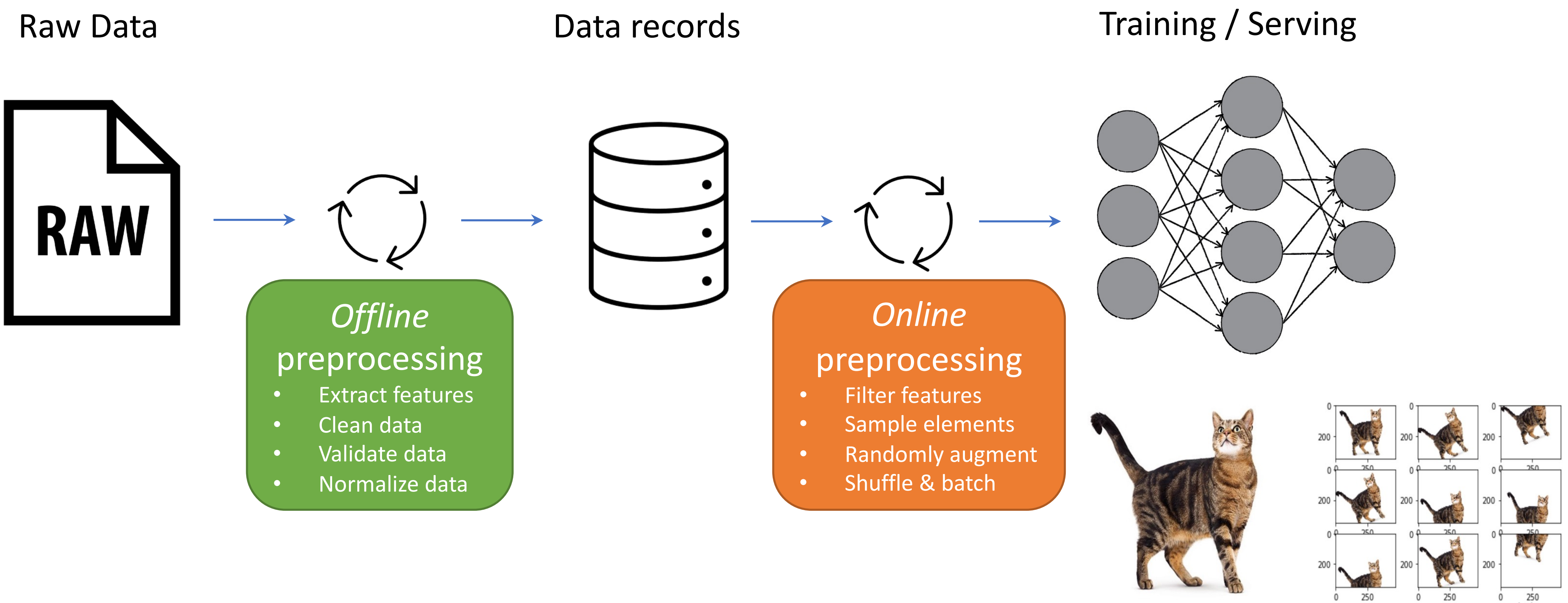
Input data processing for ML

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.



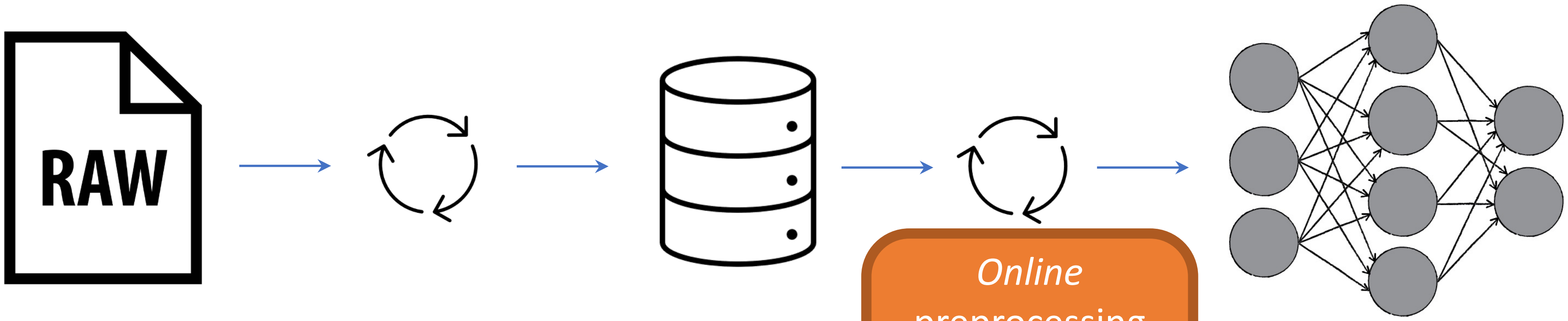
Input data processing for ML

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

Raw Data

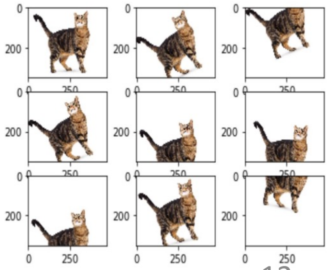
Data records

Training / Serving

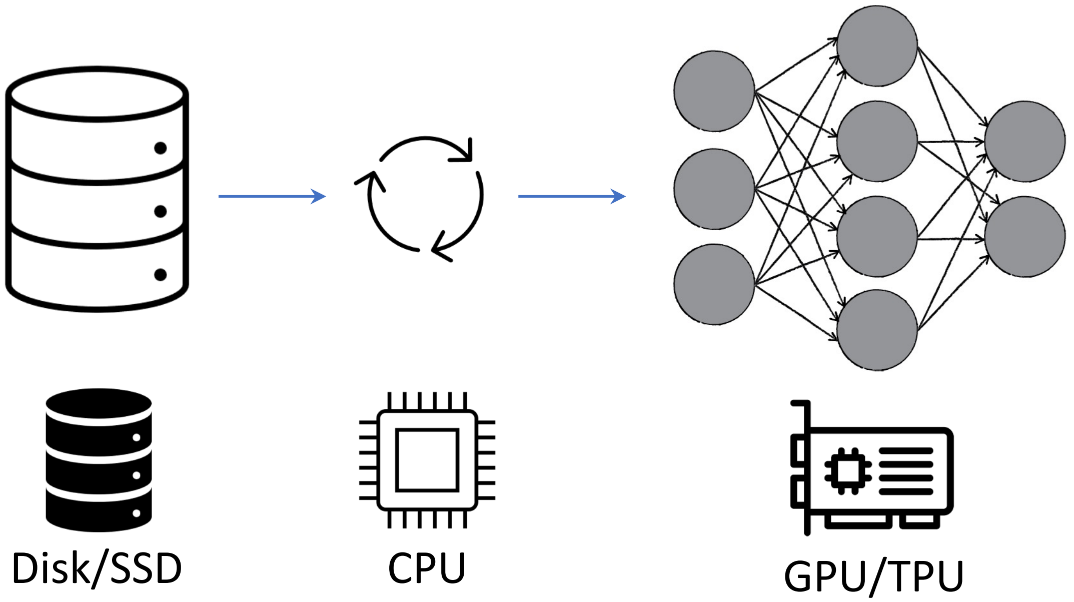


Online preprocessing
- Filter features
- Sample elements
- Randomly augment
- Shuffle & batch

"Last mile" data processing

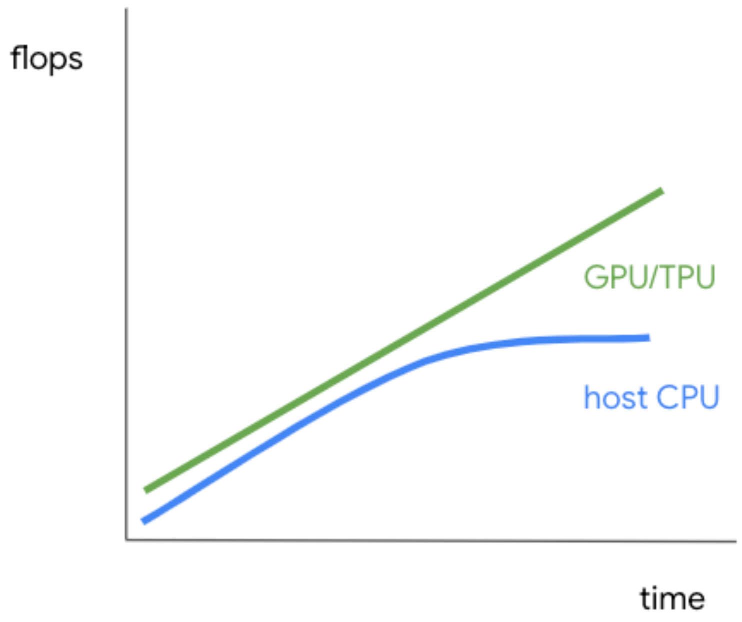


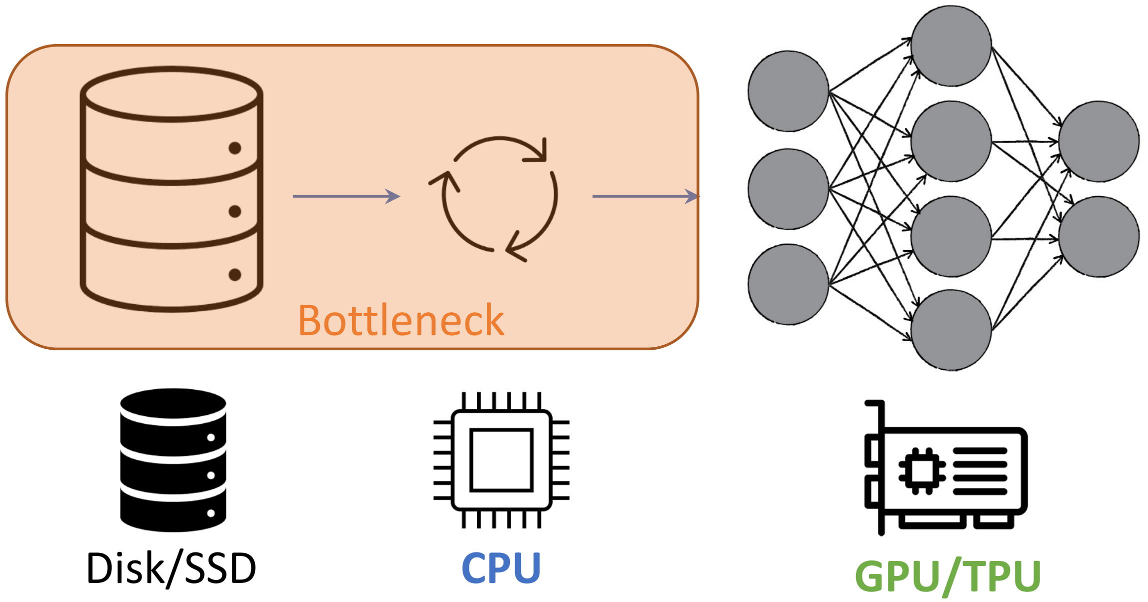
Input processing impacts training time & cost



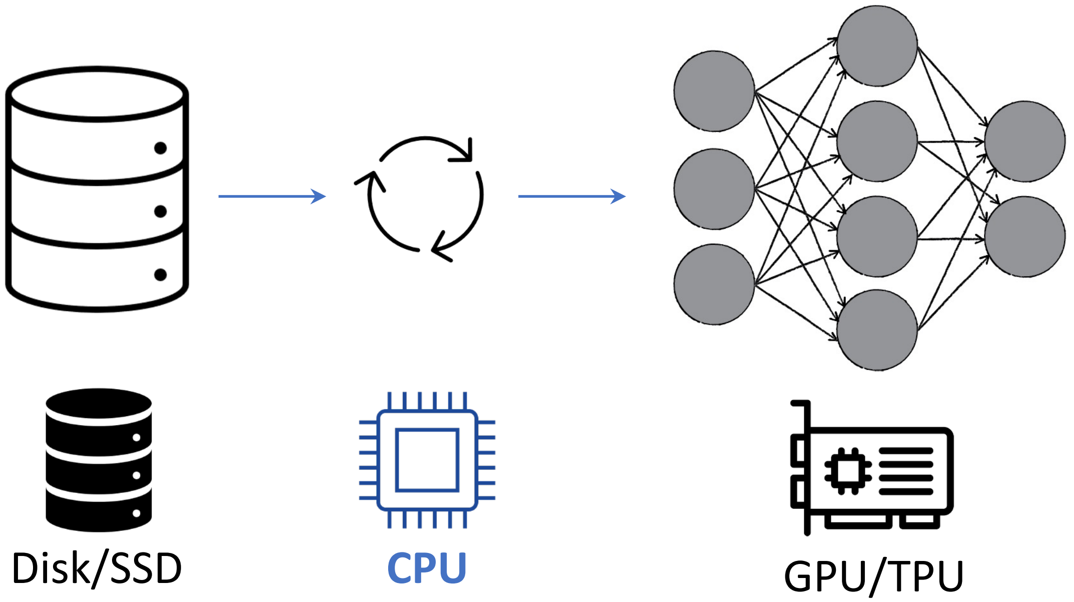
Input processing impacts training time & cost

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



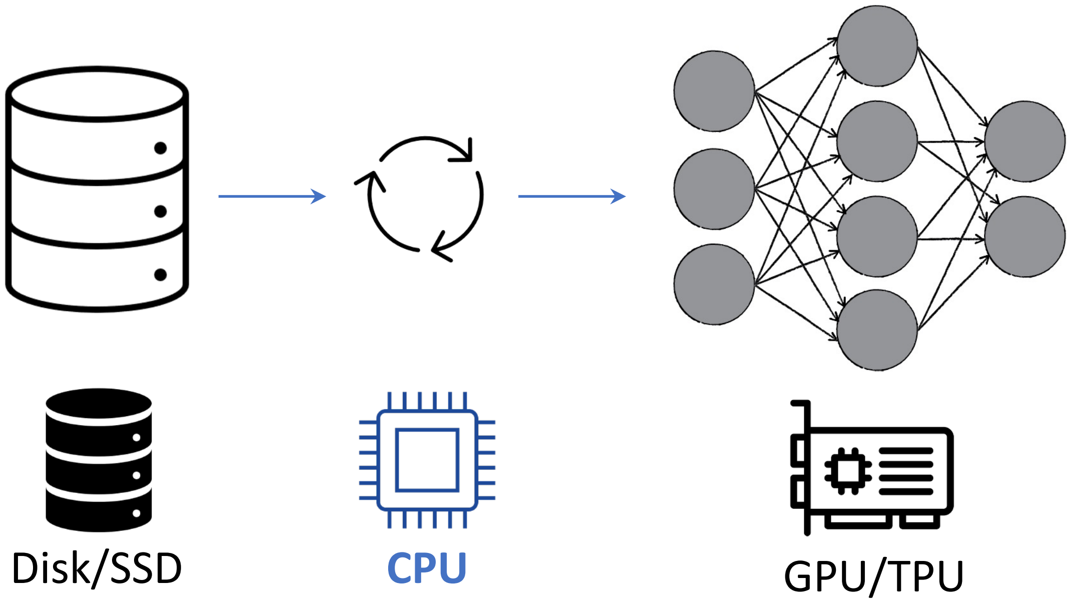


Input processing consumes high CPU/energy



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.

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

```
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

```
import tensorflow as tf

def preprocess(record):
  ...

dataset = tf.data.T
dataset = dataset.ma
dataset = dataset.ba  ch(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

```
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):
  ...


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

`tf.data.AUTOTUNE`

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

```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

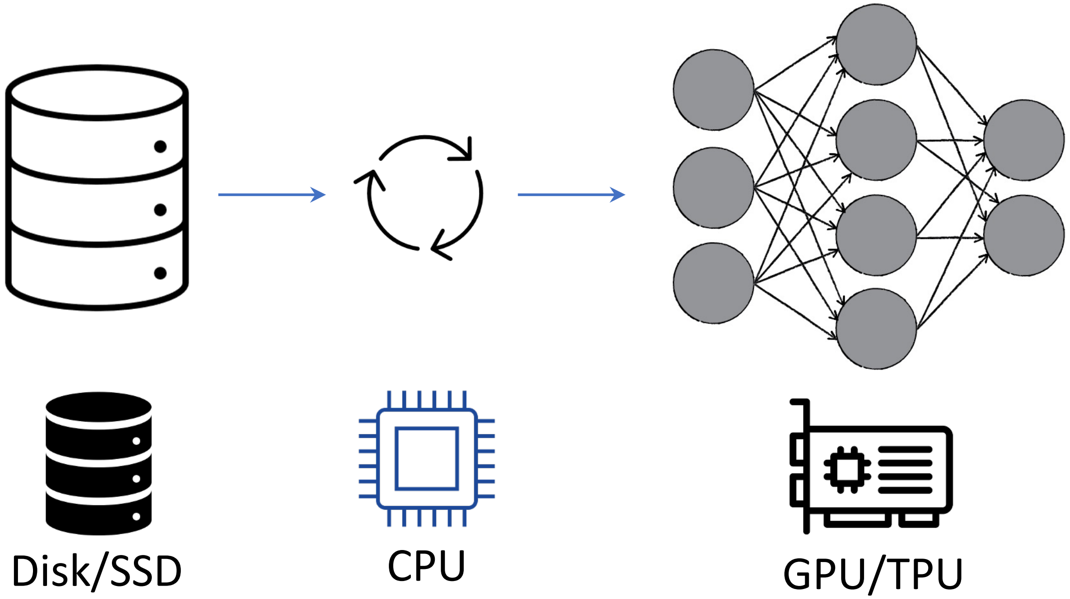
`tf.data.AUTOTUNE`

Autotuning can also be cast as an **integer linear program.**

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

Autotuning optimizes the throughput of the input pipeline given a fixed amount of CPU/memory on the host training machine.

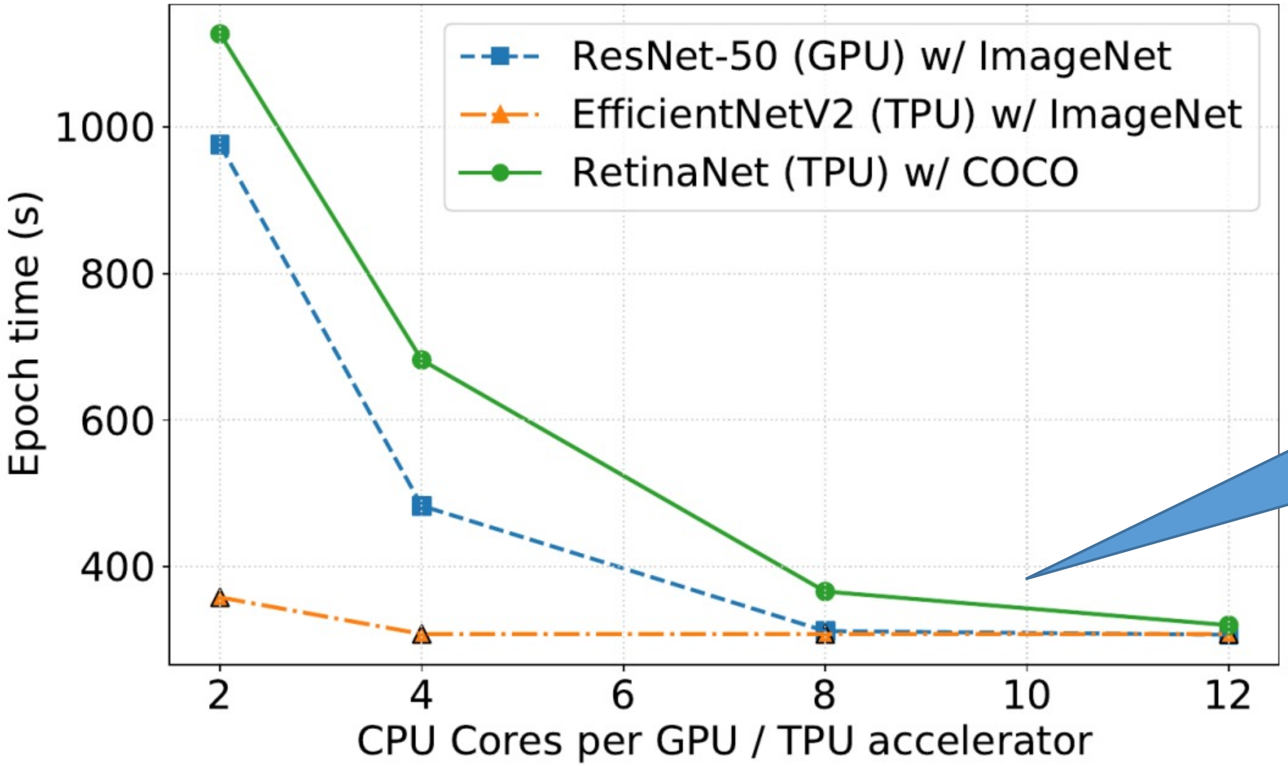
What if we don’t have enough host resources to avoid data stalls?



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

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

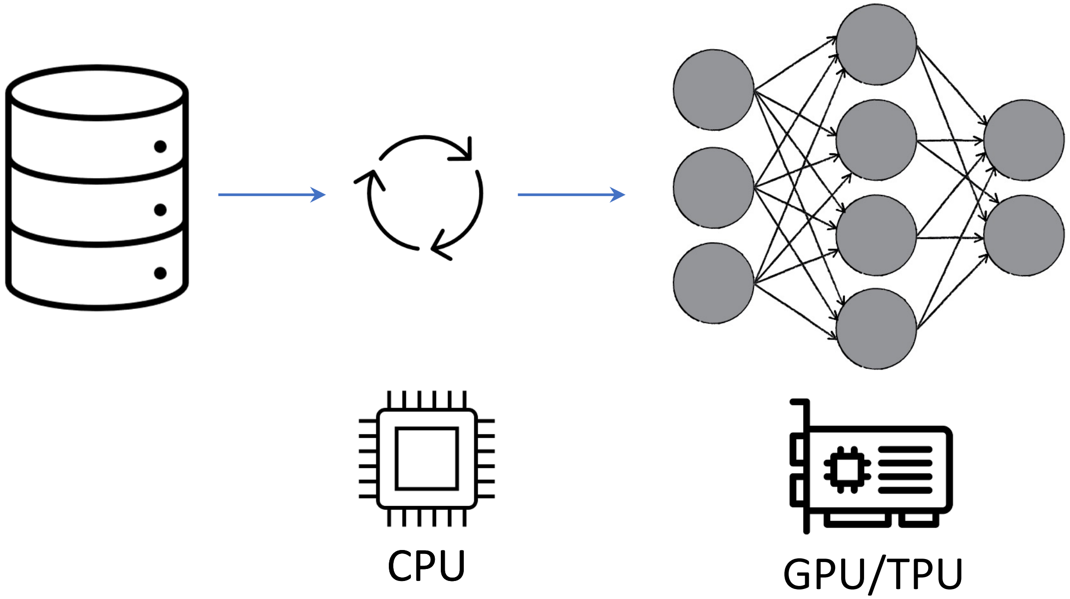
→ Ideal resource allocation depends on the model and input pipeline



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

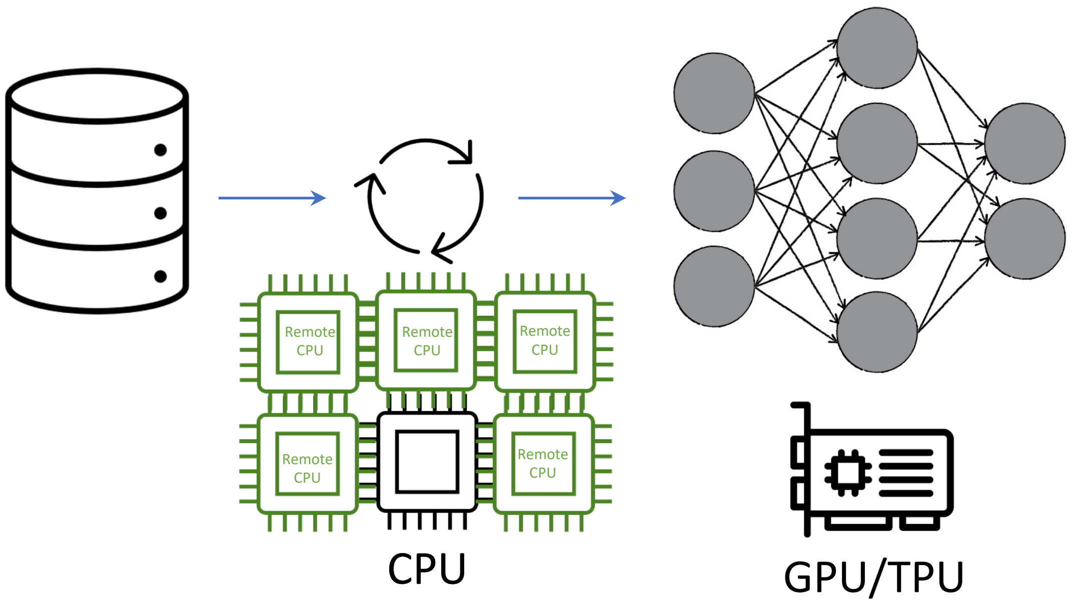
Solution: disaggregate input data processing

- Independently scale resources for input data processing & model training



Solution: disaggregate input data processing

- Independently scale resources for input data processing & model training



Solution: disaggregate input 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:

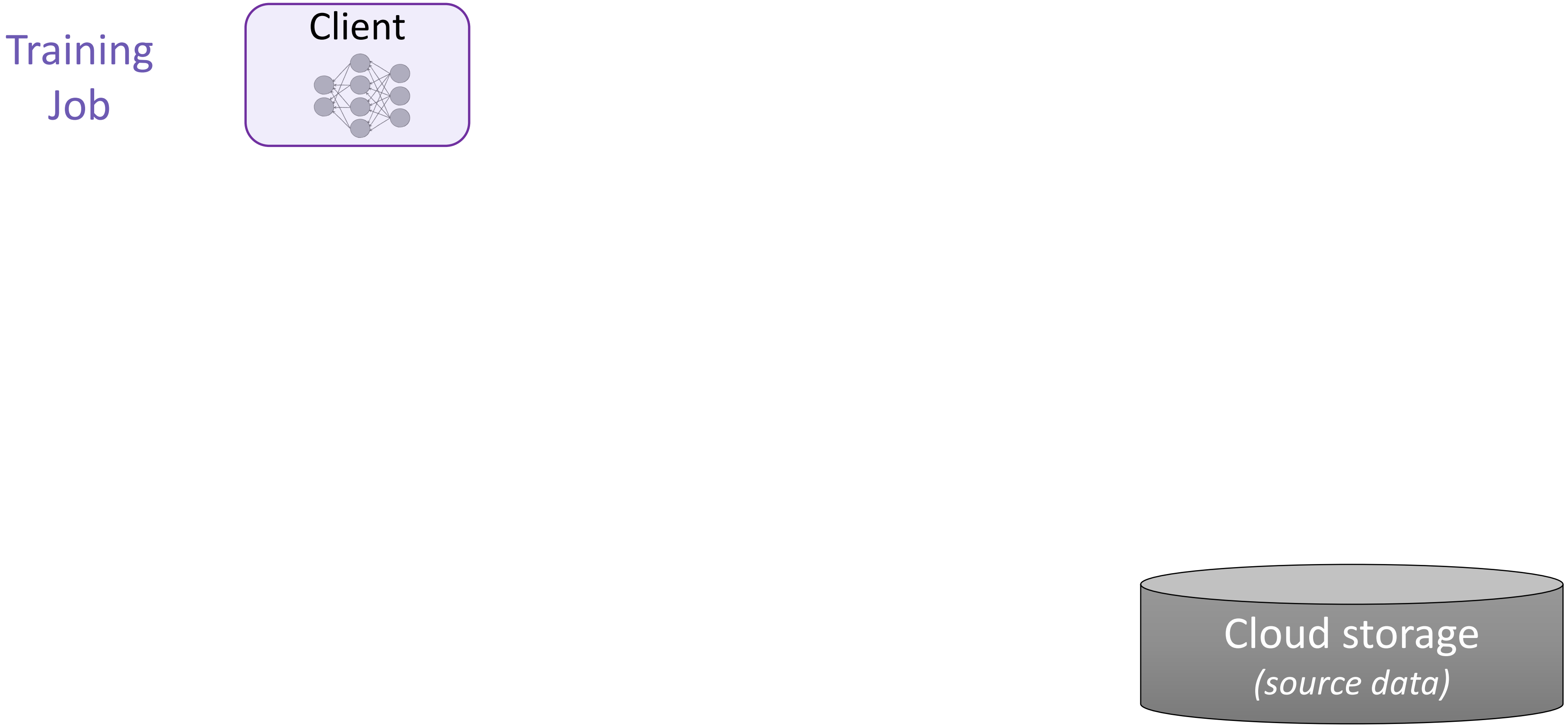
Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, Parik Pol. 2022. Understanding Data Storage and Ingestion for Large-Scale 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.

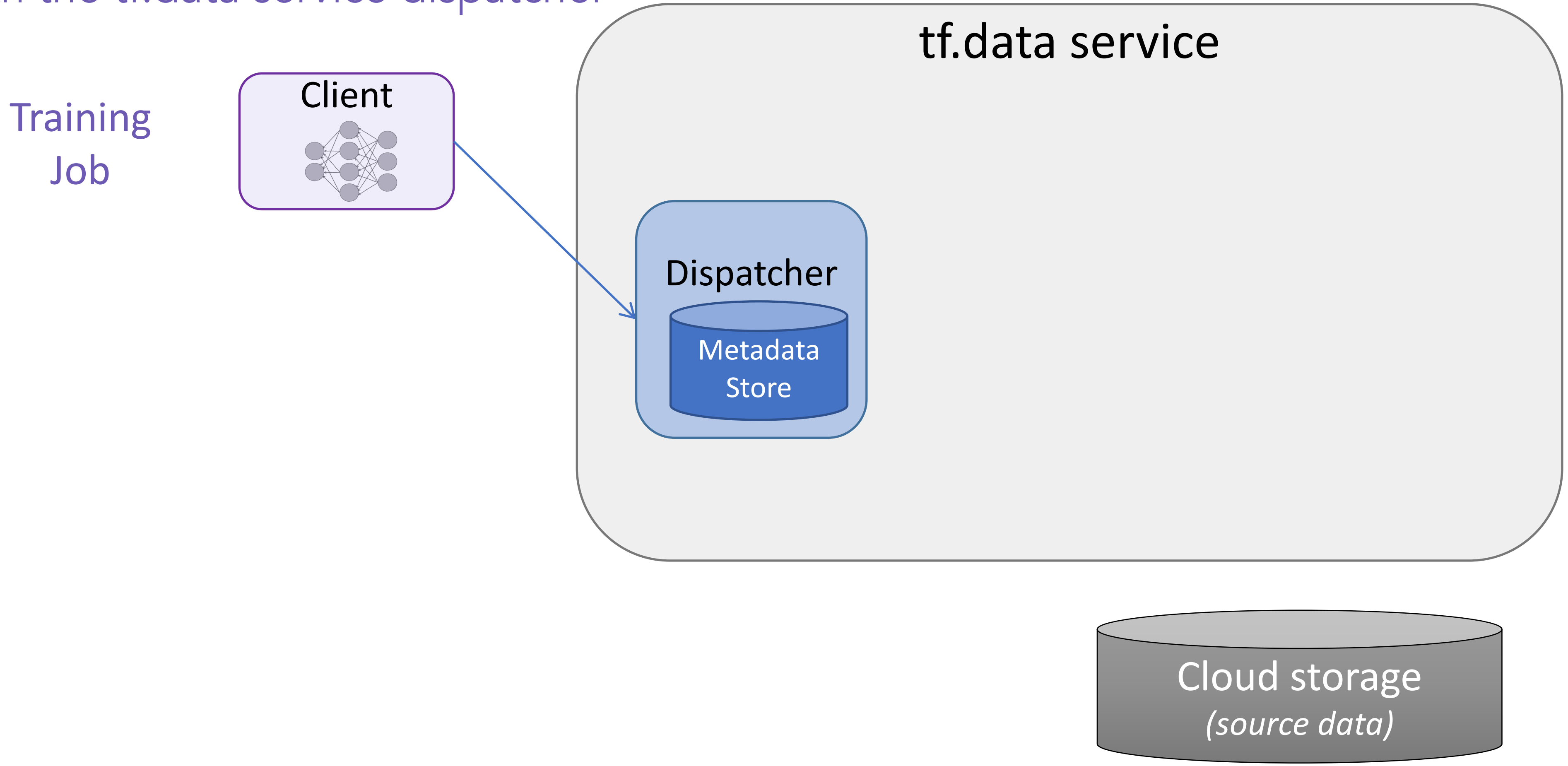
DSAs are increasingly deployed in immense scale-out systems to train increasingly-complex and computationally-demanding DNNs using massive datasets. For example, the latest MLPerf Training round (v1.1) [56] contains submissions from Azure and NVIDIA using 2048 and 4320 A100 GPUs, respectively, whereas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

tf.data service: disagg ML data processing



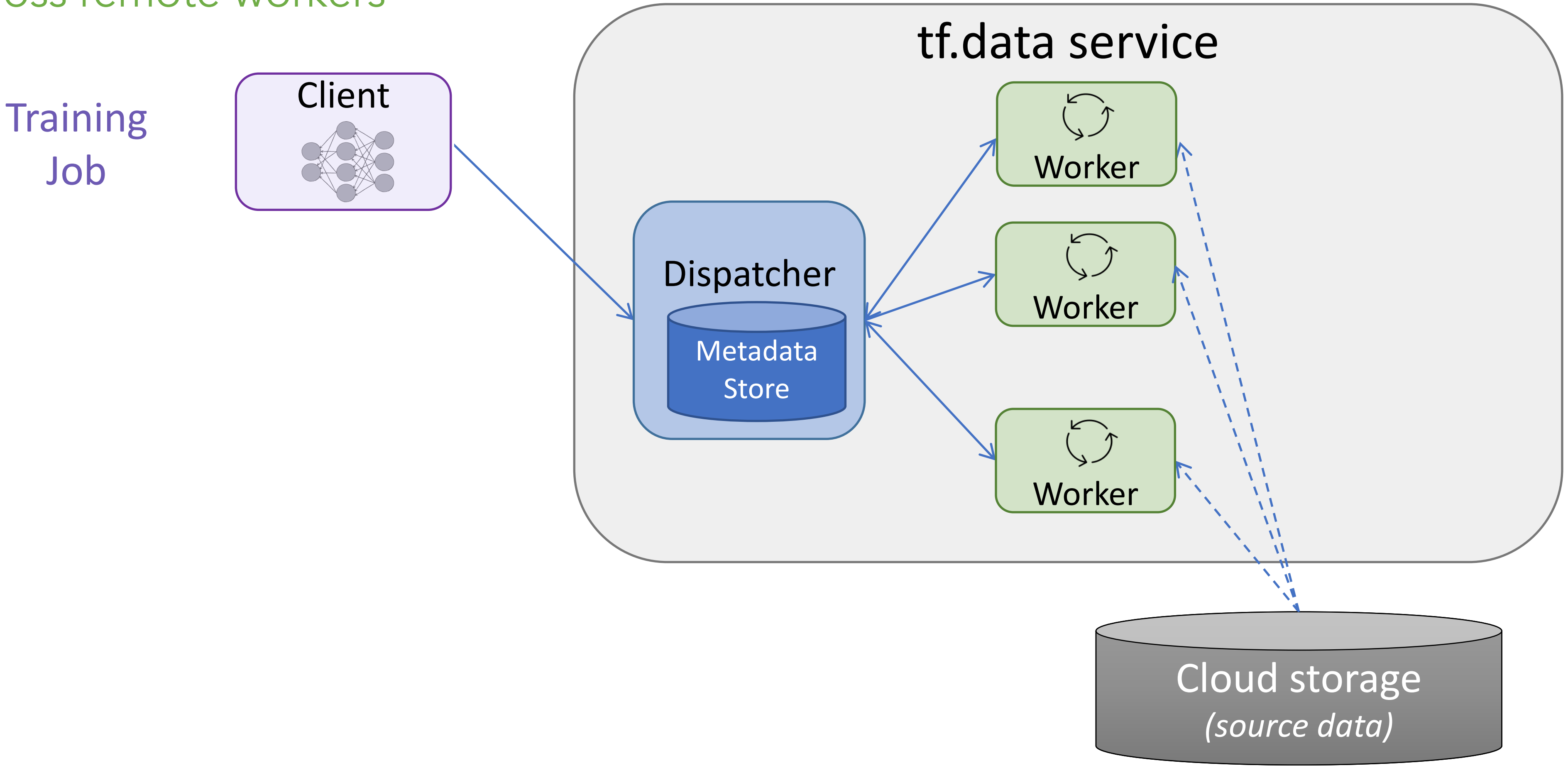
tf.data service: disagg ML data processing

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



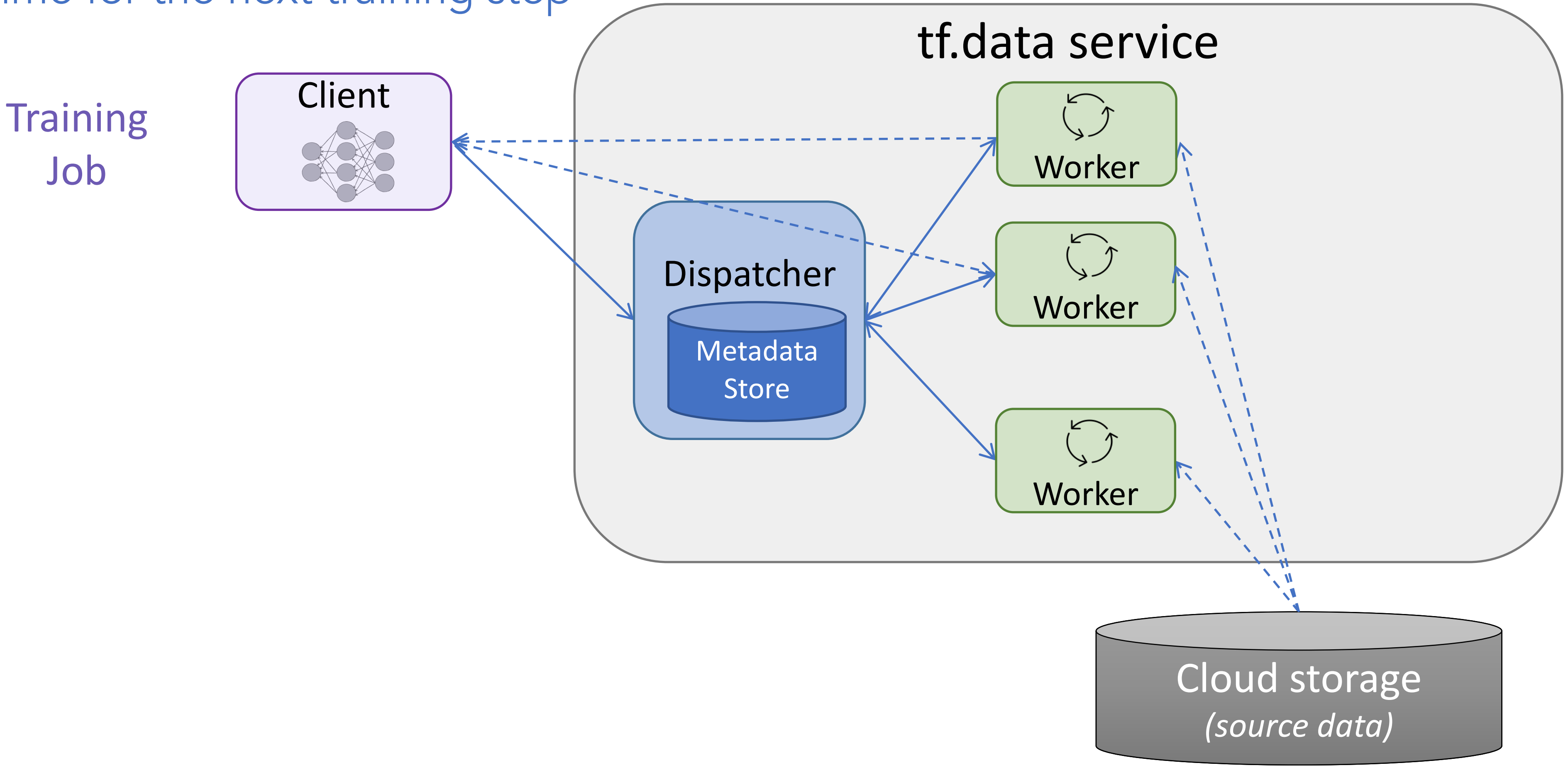
tf.data service: disagg ML data processing

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)
```

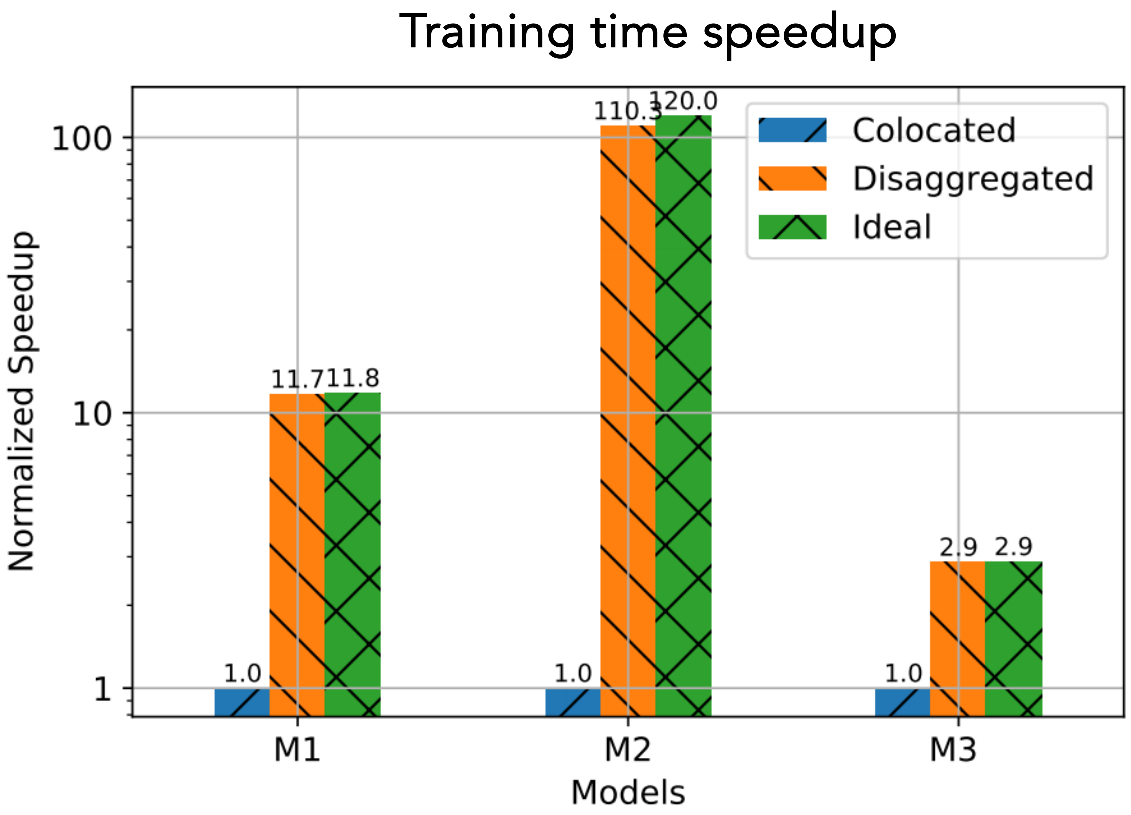
register input pipeline with dispatcher

Benefits of disaggregated ML data processing

Remove input bottlenecks

Benefits of disaggregated ML data processing

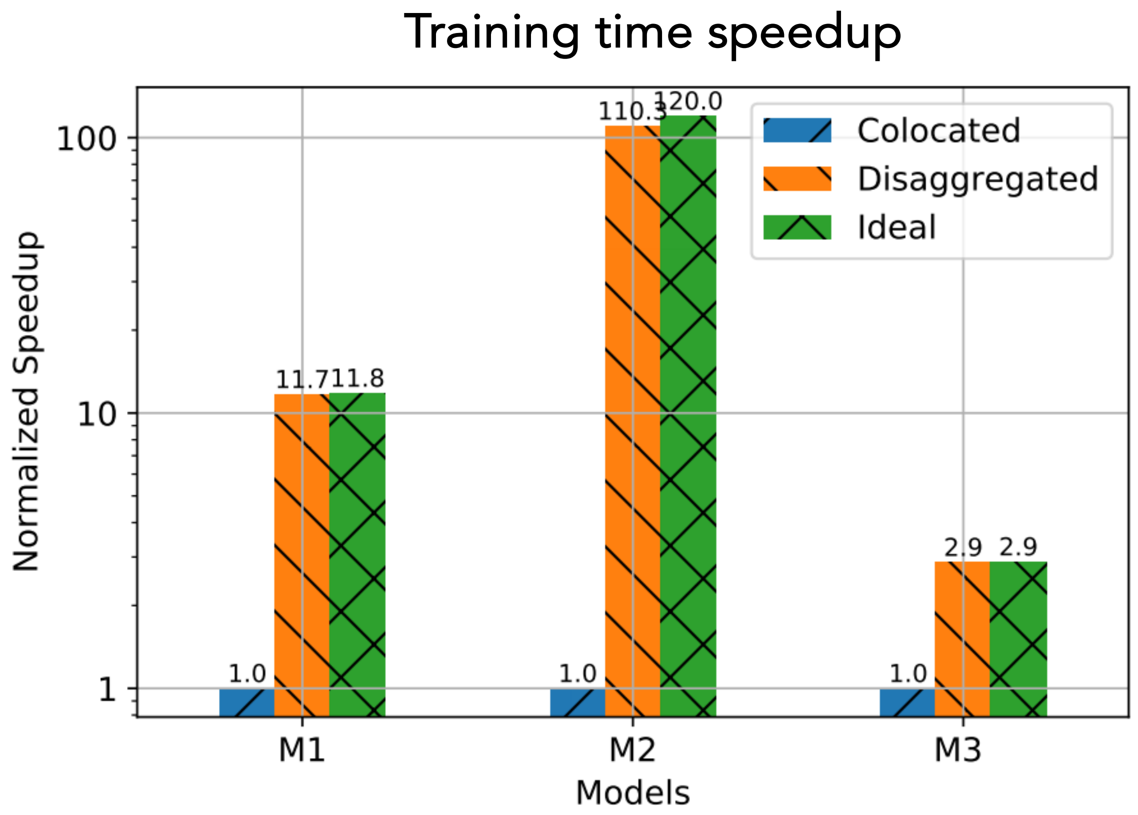
Remove input bottlenecks → up to **110x** speedup

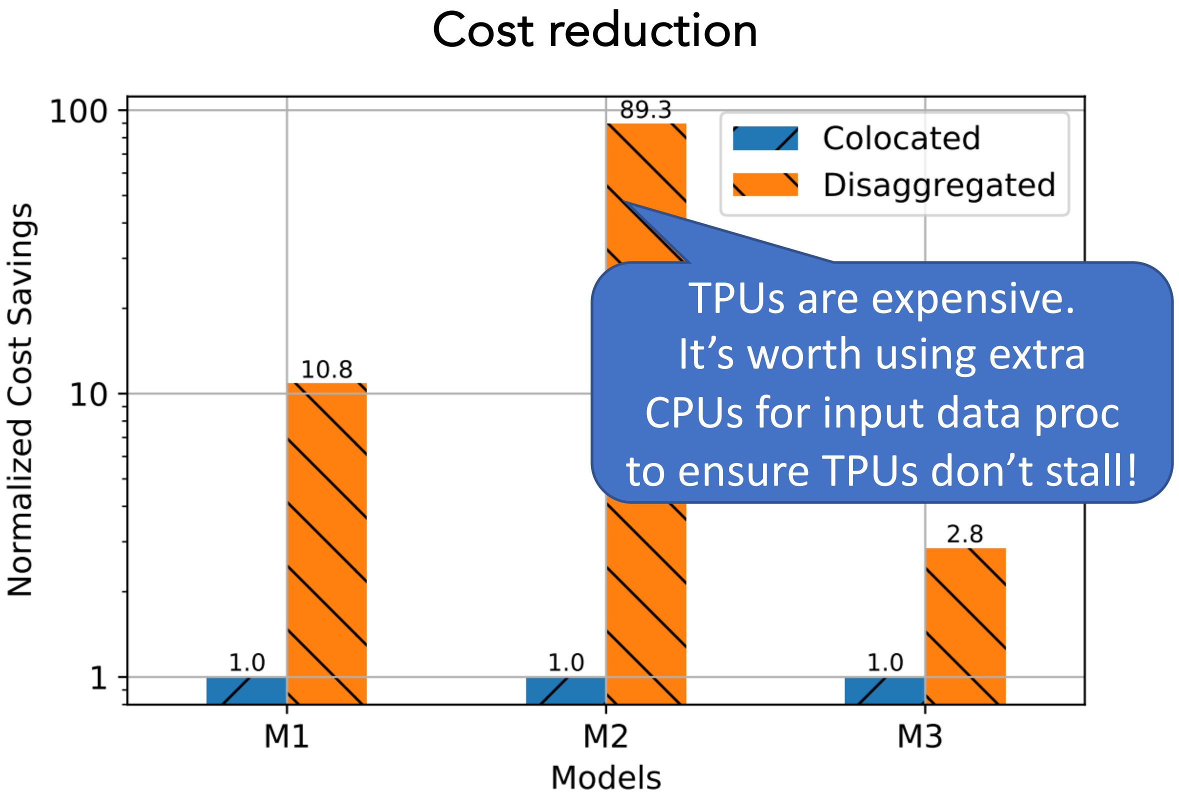


Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiri Simsa, Chandu Thekkath. *[tf..data service: A case for disaggregating ML input data processing]*, SoCC 2023.

Benefits of disaggregated ML data processing

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

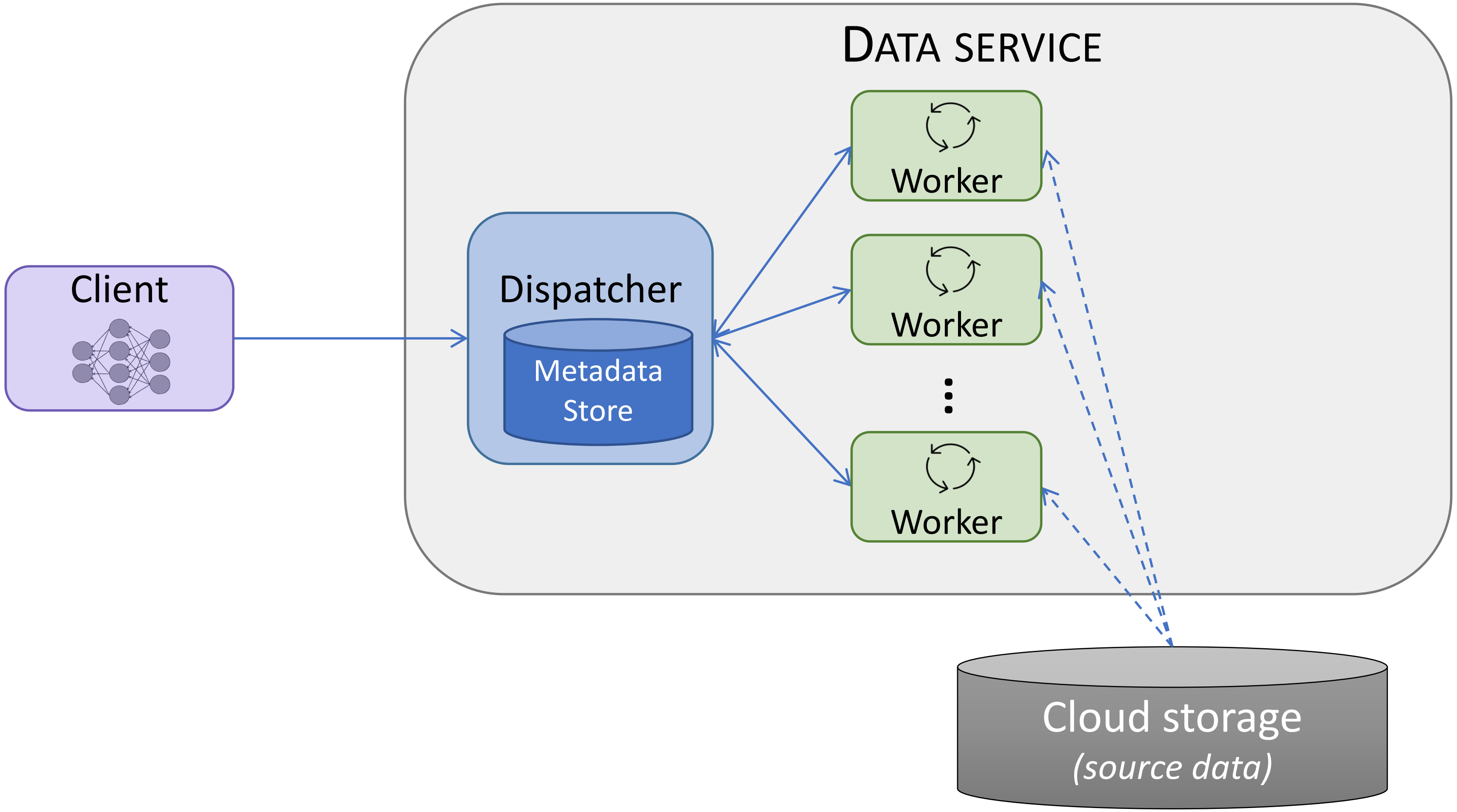




Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiri Simsa, Chandu Thekkath. [tf..data service: A case for disaggregating ML input data processing](), SoCC 2023.

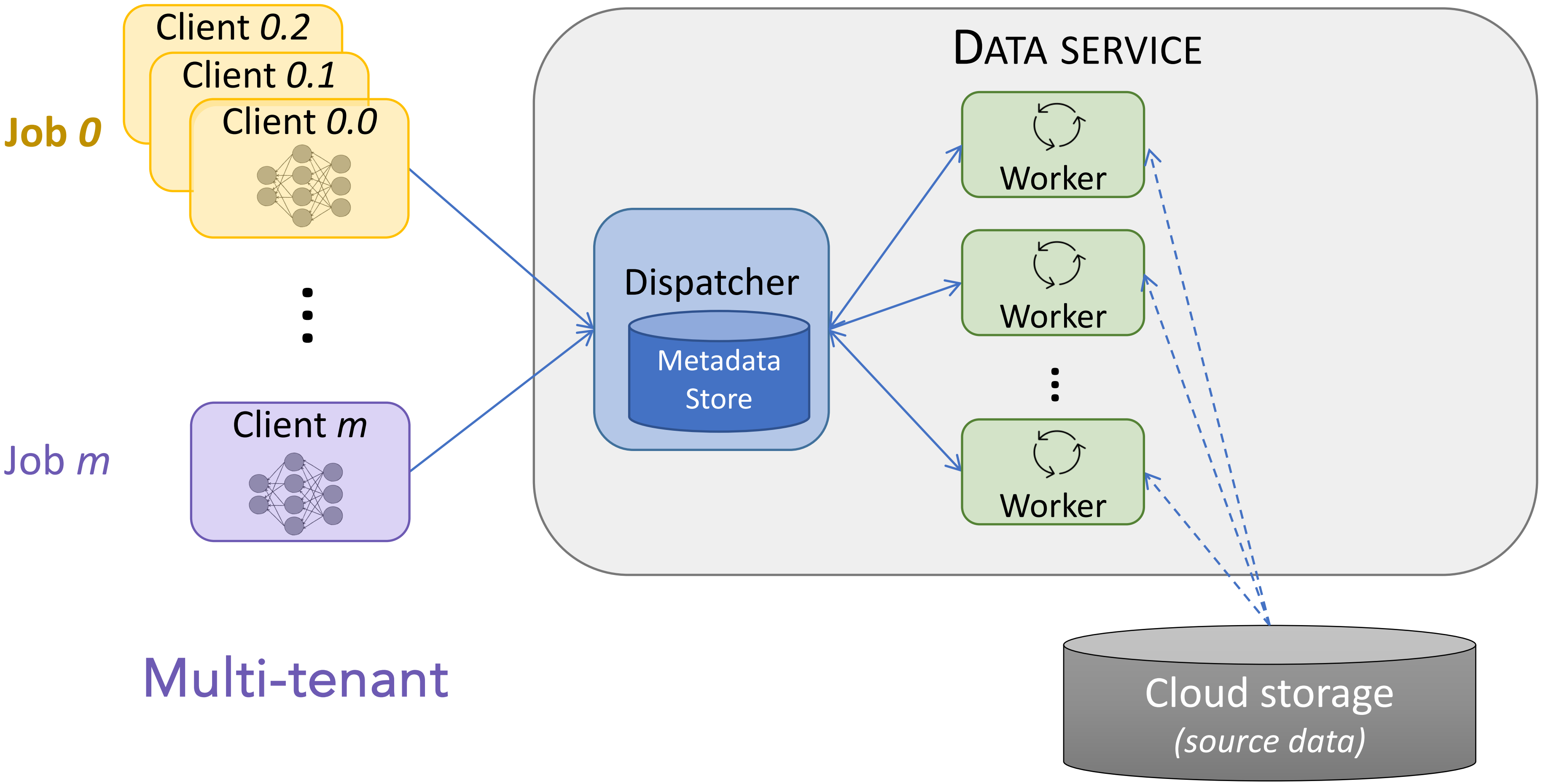
ML data processing **as a service**

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



ML data processing **as a service**

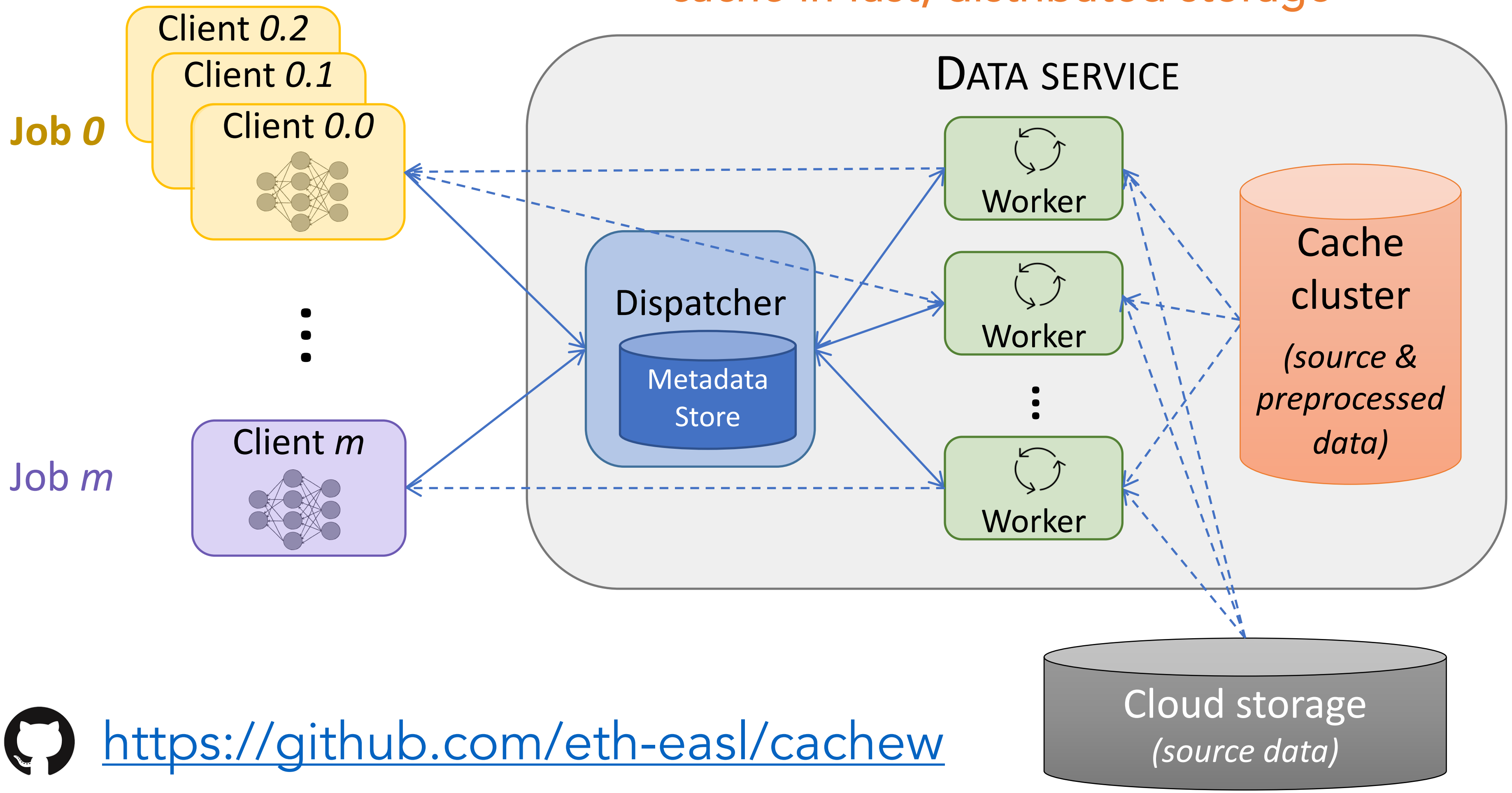
Can we leverage a global view of data processing **across jobs**?



Multi-tenant

Cachew: ML data processing as a service

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



Dan Graur

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

Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandu Thekkath, Ana Klimovic. *Cachew: ML Input Data Processing as a Service*, USENIX ATC 2022.

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

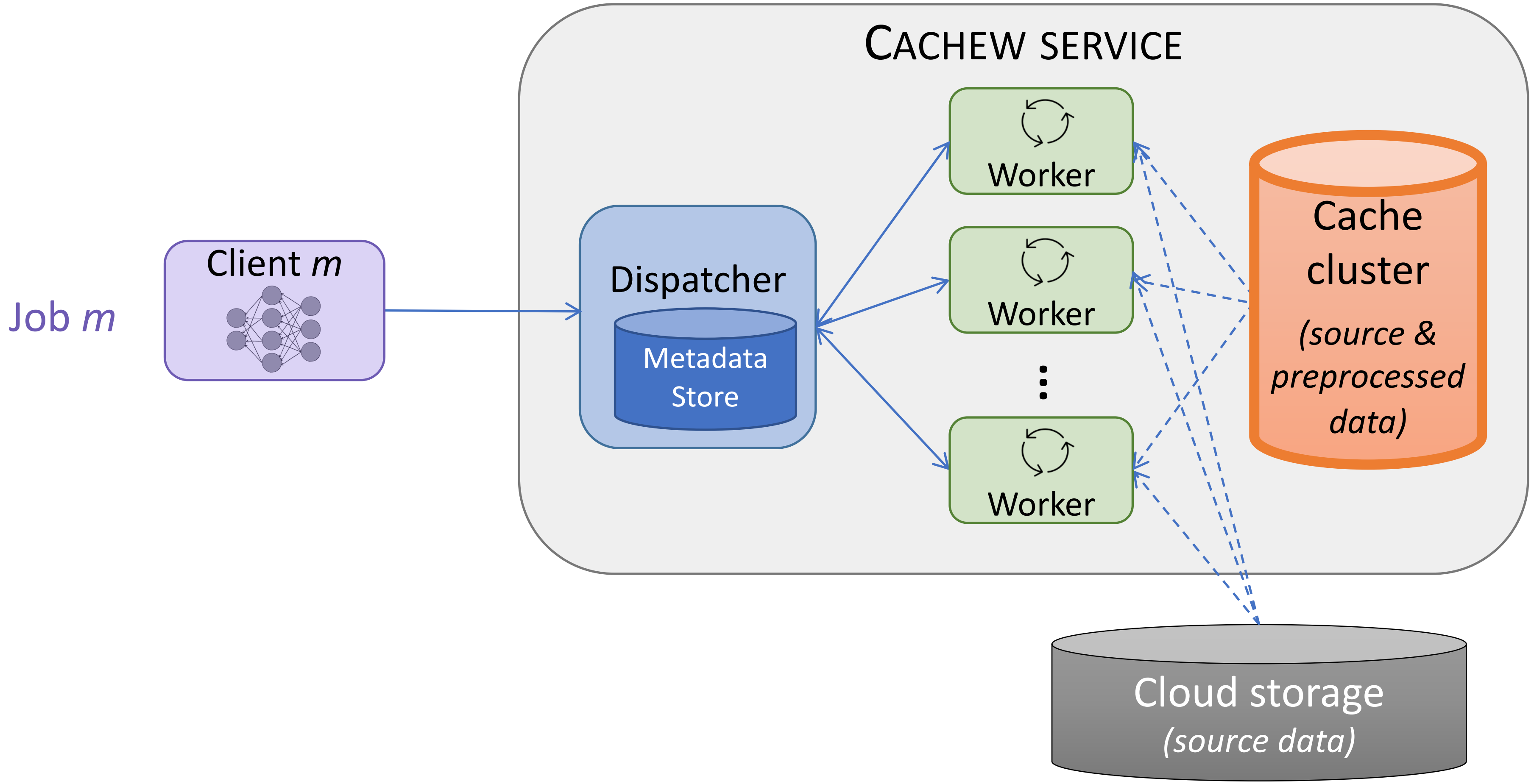
1. How to efficiently autoscale resources for input data processing?
2. How/when to efficiently cache and re-use (transformed) datasets?

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)

Autocaching policy

How/when to efficiently **cache and re-use** (transformed) datasets?



```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

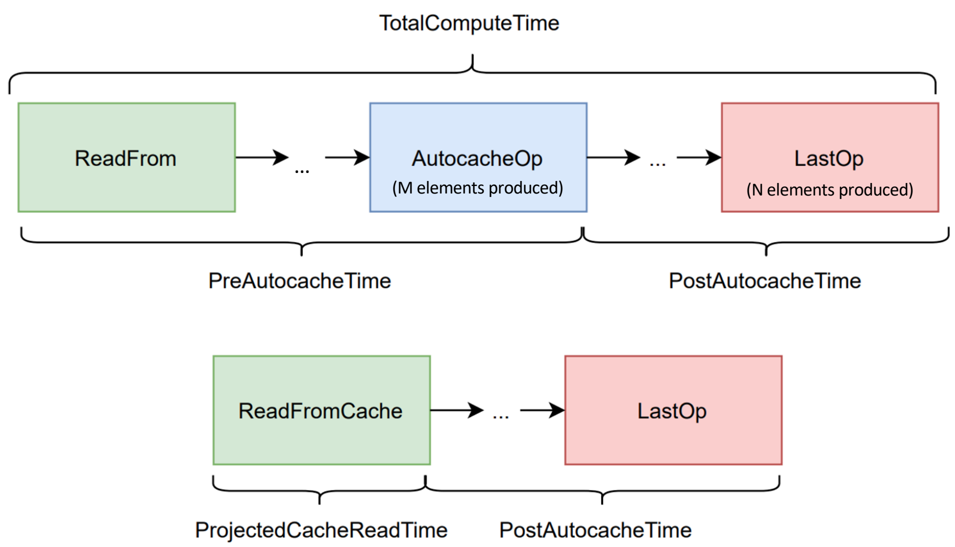
```python
import tensorflow as tf

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

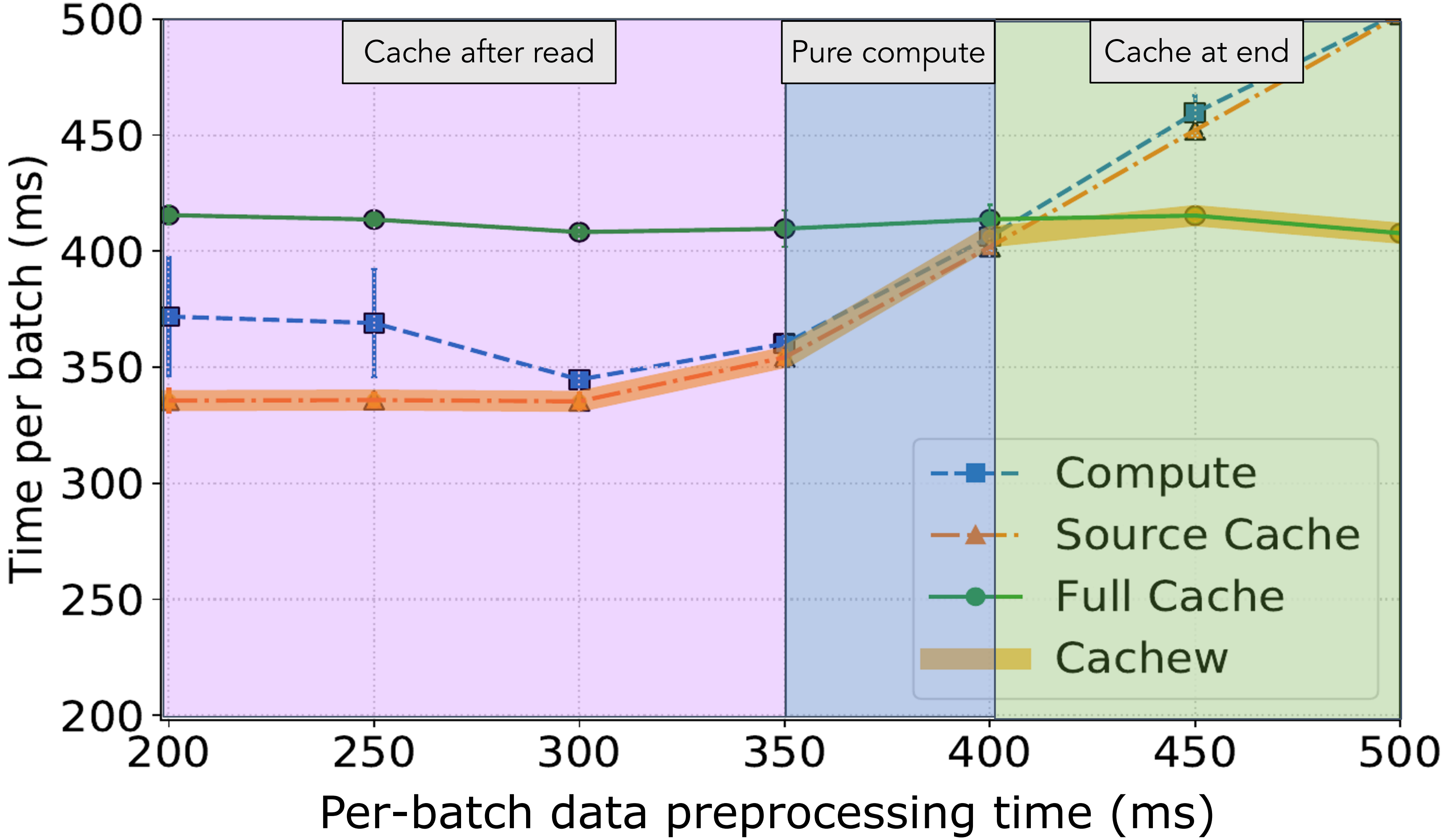
- During first epoch, at each **autocache** op, infer *compute* vs. *cache* read throughput:



- 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

Autocaching policy evaluation

Measure batch time with synthetic input data pipeline that augments source data by 2.5X.



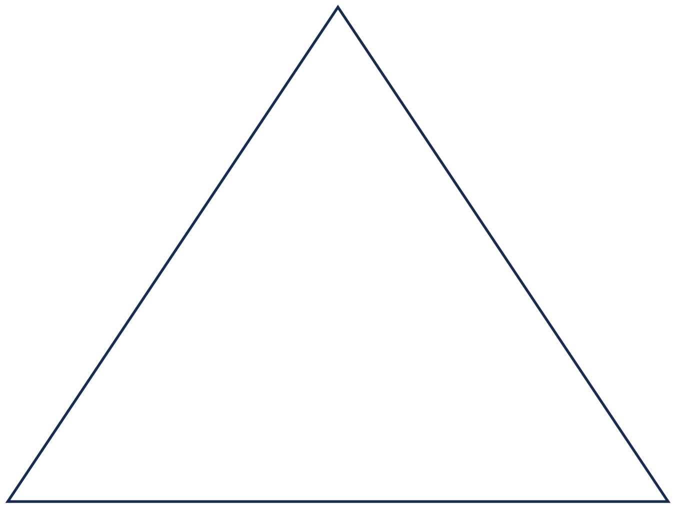
Future directions for ML data services

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

How can we reduce the cost of ML?

Resource efficiency → maximize GPU/TPU utilization



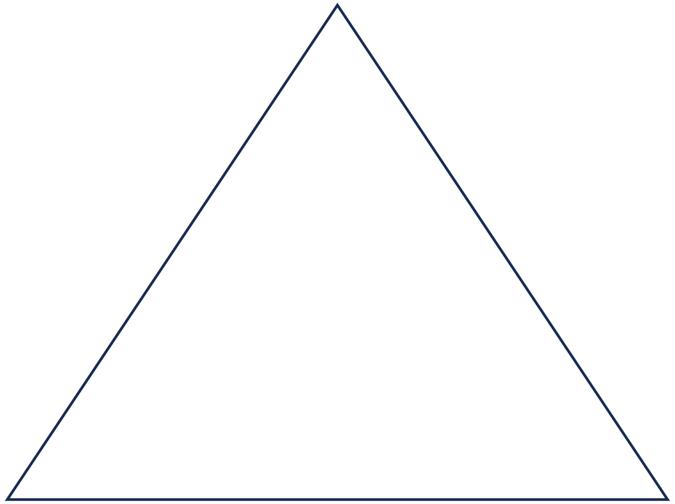
Data efficiency

→ train on the most important data

Model efficiency

How can we reduce the cost of ML?

Resource efficiency → maximize GPU/TPU utilization

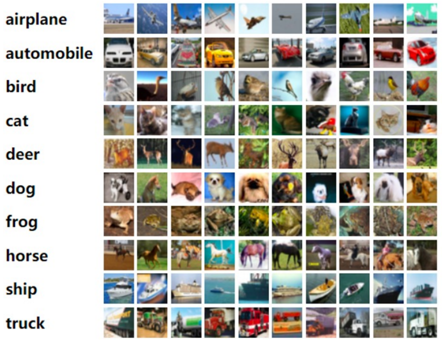


Data efficiency

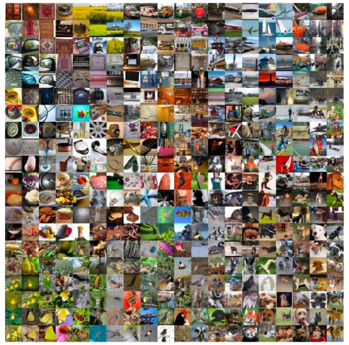
→ train on the most important data

Model efficiency

MNIST



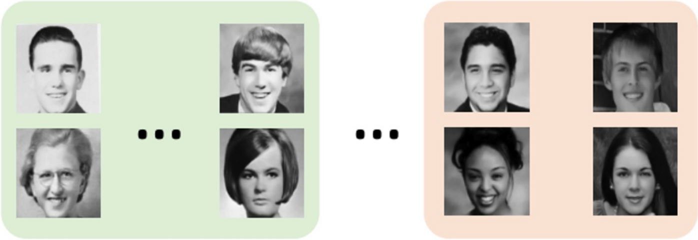
CIFAR



ImageNet

Real datasets are often *dynamic*

More data collected



Data *shifts*

Art. 17 GDPR

Right to erasure ('right to be forgotten')

Data needs to be *deleted*

→ ML models need to be updated!

How much can retraining help?

Example: online recommendations for GrubHub food delivery in 2021

Retraining Method	Purchase Through Rate Increase
No Retraining	0
Weekly Retraining	+2.5%
Daily Retraining	+20.3%

Online Learning for Recommendations for GrubHub, Alex Egg, 2021.

The cost of model retraining

Is proportional to:

~ How often retrain

~ How many data samples use for training

How to update models cost-efficiently?

- When to trigger retraining?

- What data to train on?

Today, ML practitioners
decide ad-hoc!

modyn : platform for ML on dynamic data

- Pluggable training triggering + data selection policies
- Manages dynamic datasets (and associated metadata) at scale, with sample-level darta selection
- Orchestrates training jobs

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

Towards a Platform and Benchmark Suite for Model Training on Dynamic Datasets. EuroMLSys'23.

Maximilian Böther

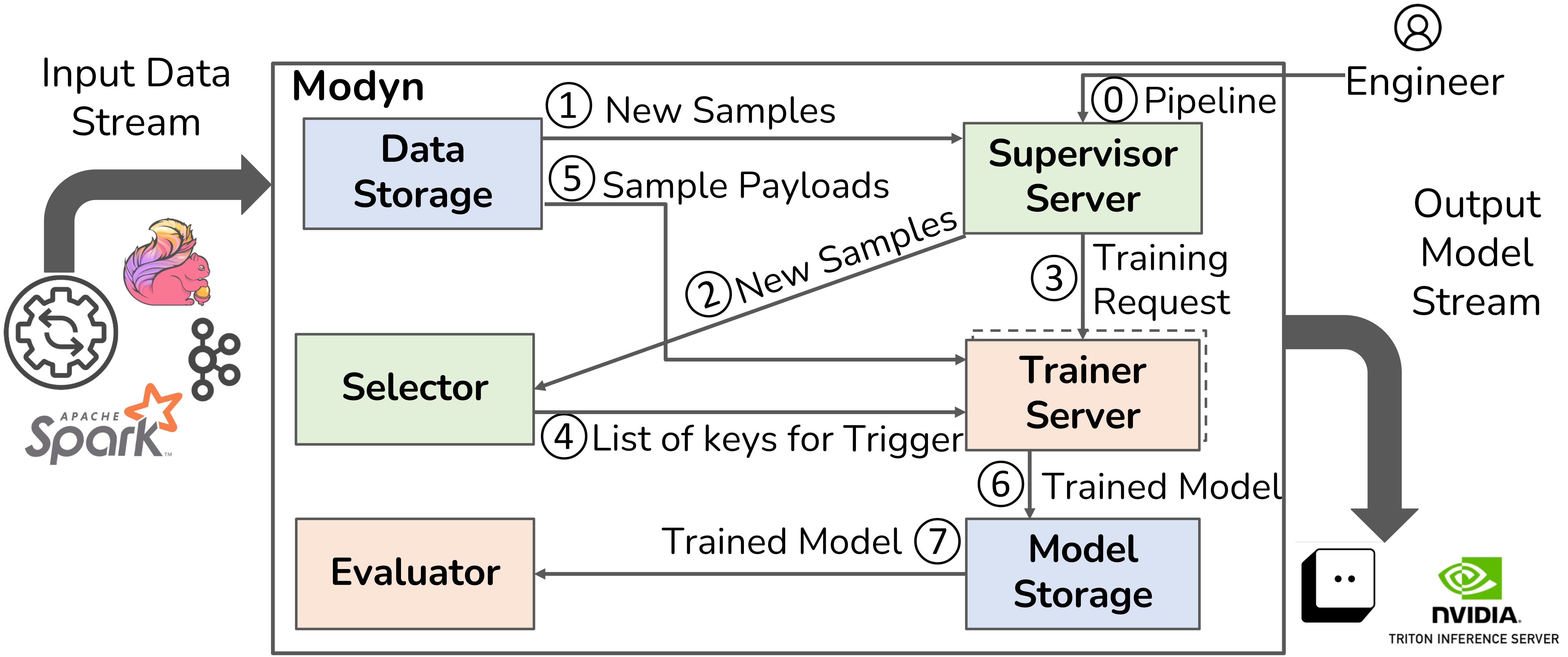
Modyn goals and challenges

Fast sample-level data selection during training

Continuous model life-cycle orchestration

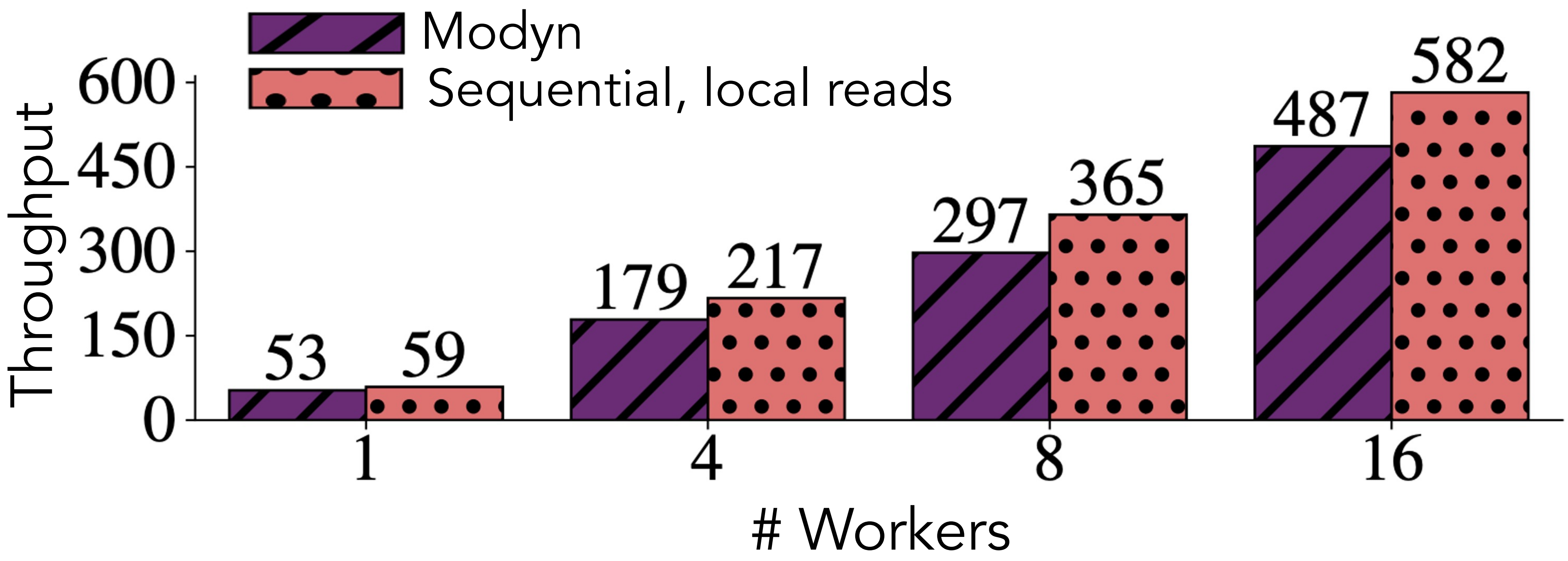
Ease of use and extensibility to ease adoption

Modyn system architecture



Training Throughput for Criteo RecSystem

Near-local throughput, even for memory-bound training.

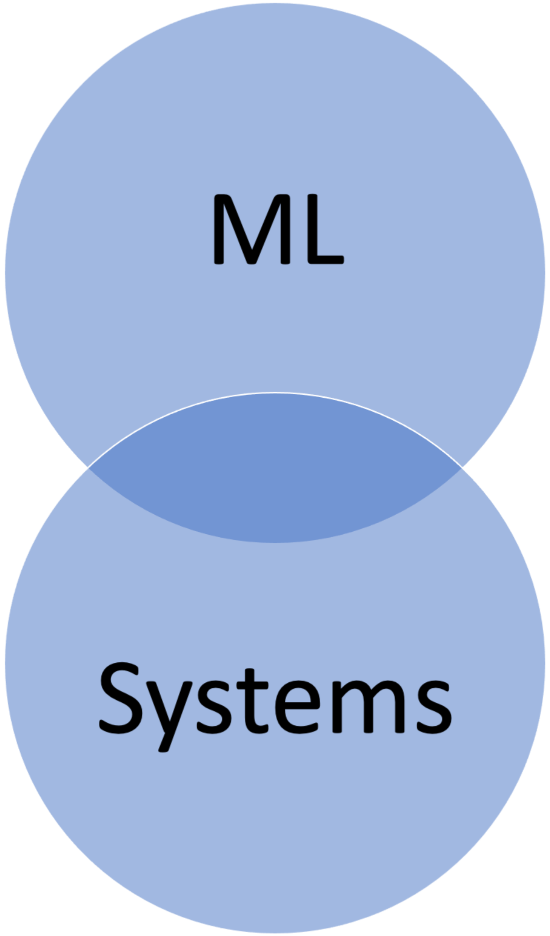


Maximilian Böther, Viktor Gsteiger, Ties Robroek, Ana Klimovic. *[Modyn: A Platform for Model Training on Dynamic Datasets With Sample-Level Data Selection](#)*. *2023*

Ongoing work on data efficiency

- Data selection policy exploration
- Model triggering policy exploration
- Exploring the interplay between the above two

- Importance-aware data placement in storage hierarchy

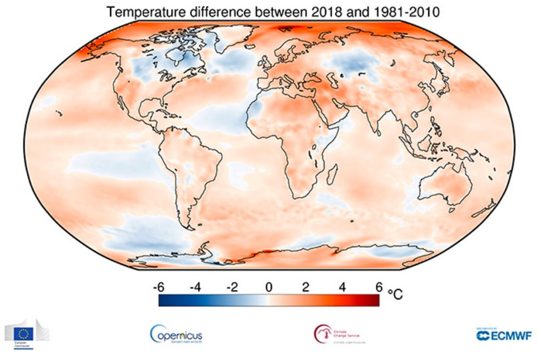


Benchmark suite for ML on dynamic data

Recommender Systems

Text Classification

Autonomous Driving



Weather Forecasting

Thanks to great collaborators ☺

Google

ETH zürich

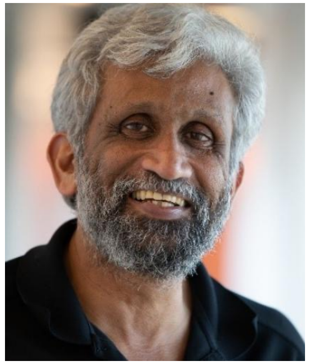

Dan Graur


Maximilian Böther


Foteini Strati


Viktor Gsteiger


Ties Robroek


Chandu Thekkath

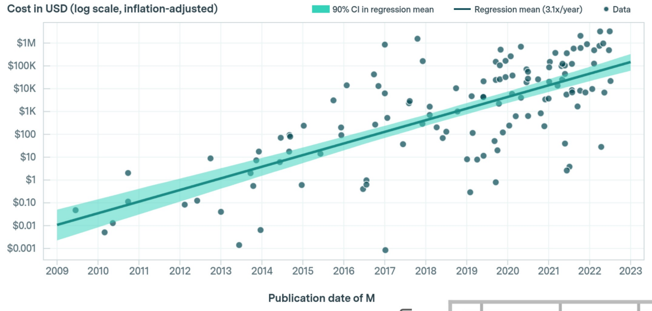

Jiří Šimša

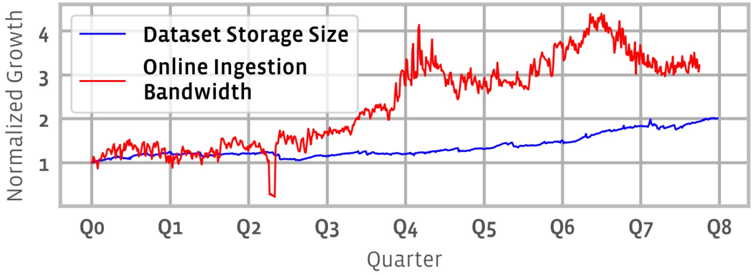

Derek Murray


Xianzhe Ma


Pinar Tözün

ML training is increasingly data hungry & expensive





How to reduce the cost of ML training?

Resource efficiency → maximize GPU/TPU utilization

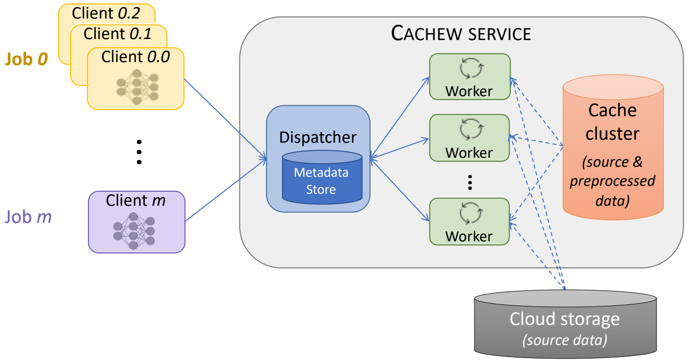
Need to optimize how we store & ingest data!

Data efficiency
→ train on the most important data

Model efficiency

Disaggregate input data processing to eliminate data stalls and maximize training throughput

Cachew



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

Train models efficiently on dynamic datasets

→ When to trigger training?
→ What data to train on?

modyn

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