TECHNICAL PAPERS • ADAPTIVE SCHEDULING FOR DISTRIBUTED AI

BatchFlow: Benefit-Aware Data Pipeline Allocation and Batch Reuse for Multi-Job Training

Patrick Watters[1], Xiaolong Ma[1], Syed Zawad[2], Jingyuan Zhang[3], Yue Cheng[4], Lei Yang[1], Feng Yan[5]

[1]University of Nevada, Reno [2]IBM Research [3]George Mason University [4]University of Virginia [5]University of Houston



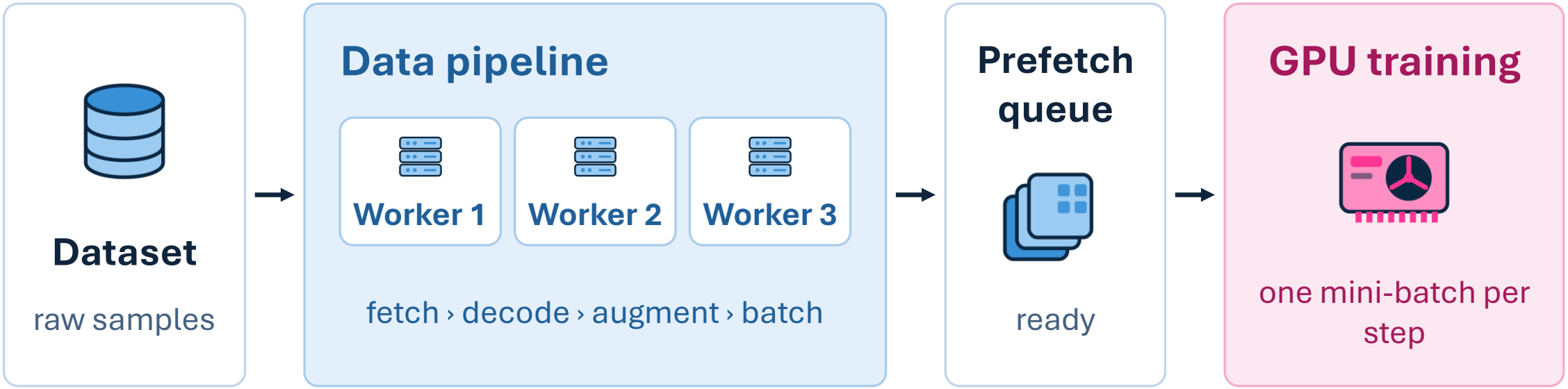
PRESENTED BY

Patrick Watters
University of Nevada, Reno

SC26 hpc unites | Chicago

Where the Data Pipeline Fits in ML Training

Parallel CPU workers prepare upcoming mini-batches while the GPU trains



When preparation falls behind, GPUs experience a data stall

Data Preparation Must Scale with GPU Demand

Training demand keeps rising

~3×

GPU compute every 2 years

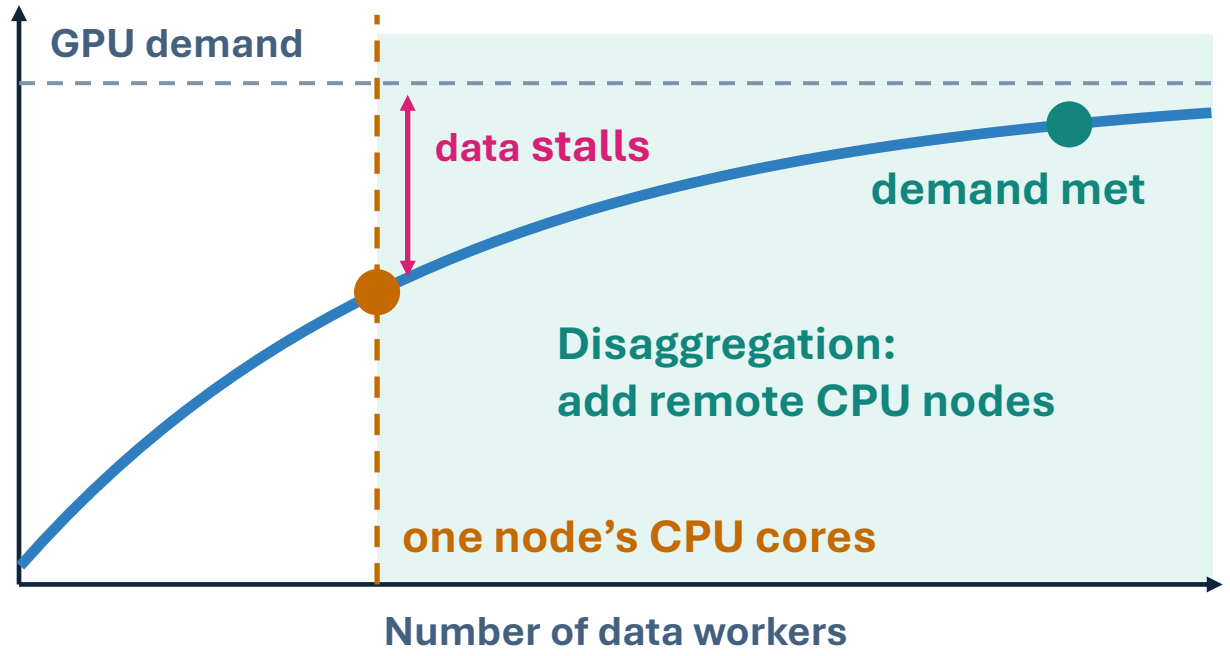
Meta Engineering · 2026

4×

training FLOPs in 12 months

Meta GEM · 2026

Training throughput



Disaggregation lets data preparation scale beyond a single training node

Jobs Share Resources but Provision Pipelines Independently

Hyperparameter tuning | **Architecture search** | **Scientific AI**

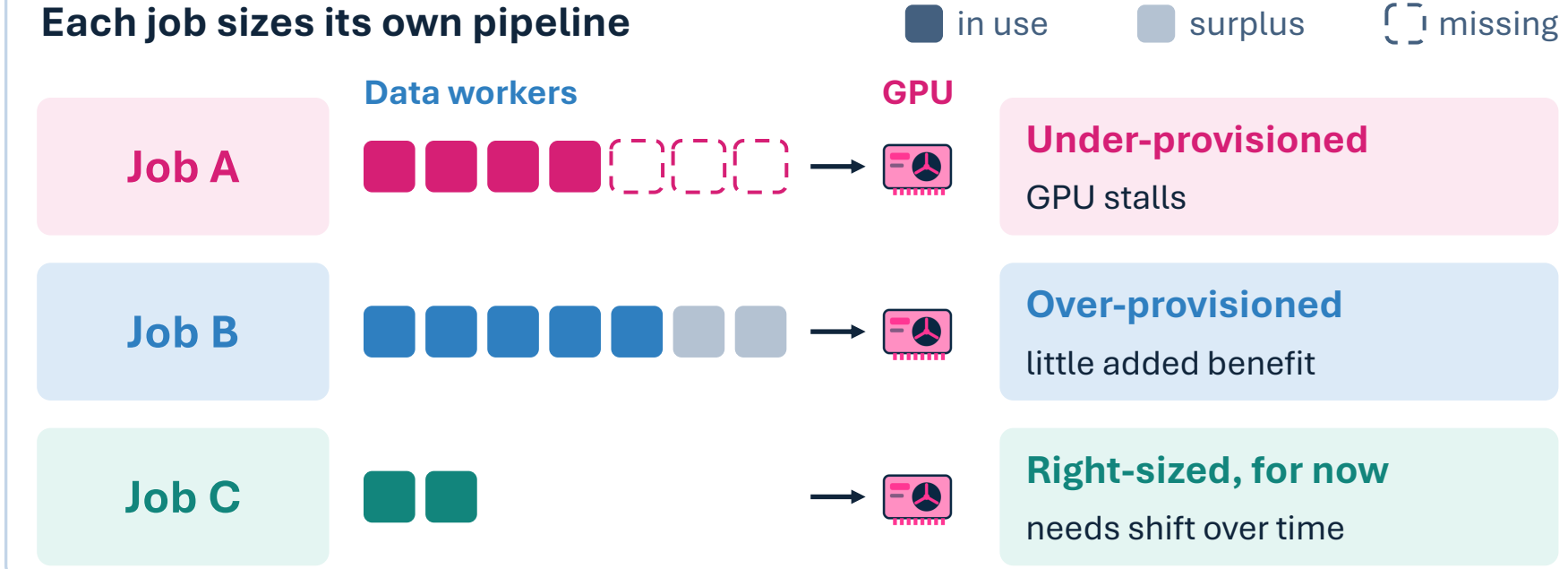
Shared resources

CPU workers

Storage

Network

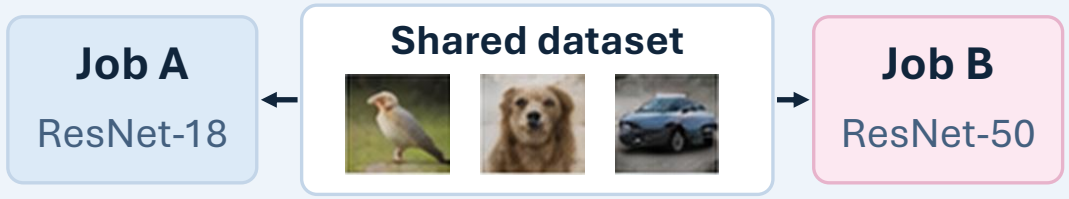
Each job sizes its own pipeline



Per-job provisioning wastes resources on some jobs and starves others

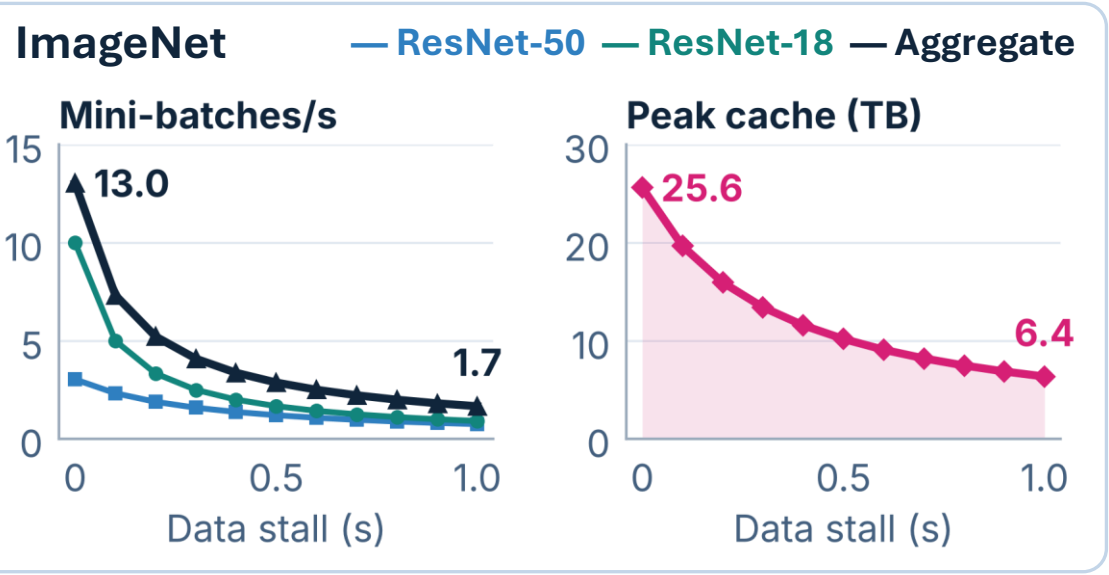
Jobs Reuse Data, but Cross-Job Reuse Is Hard

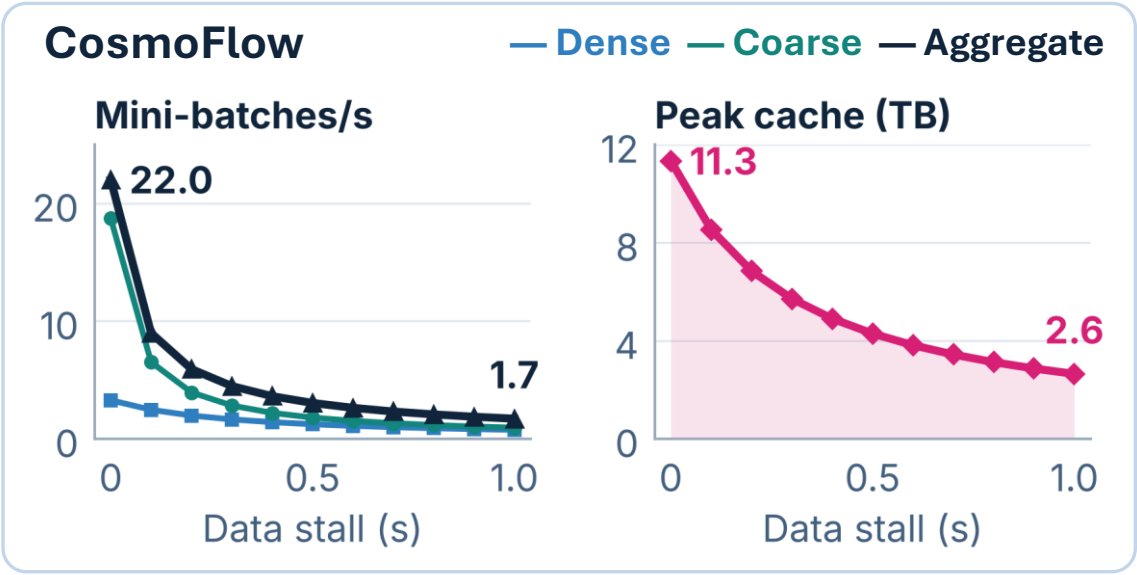
Opportunity: jobs read the same data



Challenge: jobs drift apart







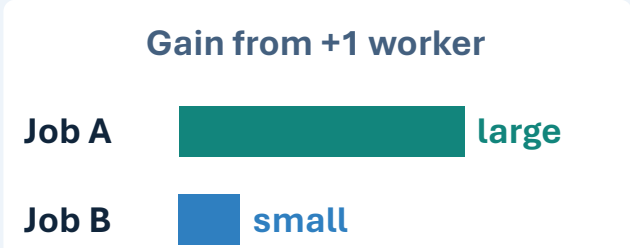
Technical Challenges

C1 Siloed pipelines



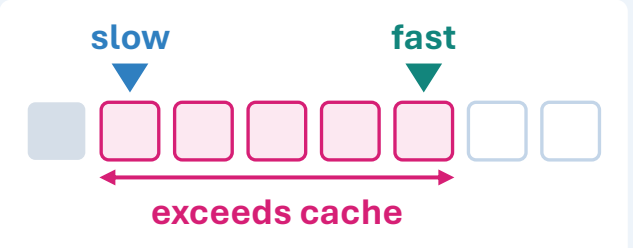
How can any worker serve any job?

C2 Uneven benefit



Where does the next worker help most?

C3 Limited cache



Which mini-batches are worth keeping?

Static, per-job pipelines cannot address these together

Problem Formulation

Choose worker allocation and cache contents to maximize total throughput

$$\max_{\mathbf{w},\,C} \sum_{j \in J} X_j(\mathbf{w}, C) \quad \text{s.t.} \quad \sum_{j \in J} w_j \le W, \quad \sum_{b \in C} s_b \le M$$

$X_j(\mathbf{w}, C)$	throughput of job j	w_j	data workers serving job j
W	total data workers	C	mini-batches kept in the cache
s_b	size of mini-batch b	M	cache capacity

Exact solution is impractical at runtime → estimate online and adapt

Key Idea: Treat Mini-Batches as a Shared Service

One shared service; workers and cache go where they help most



Shared worker pool

Any worker prepares any job's next mini-batch



Benefit-aware allocation

Workers go to jobs closest to a data stall

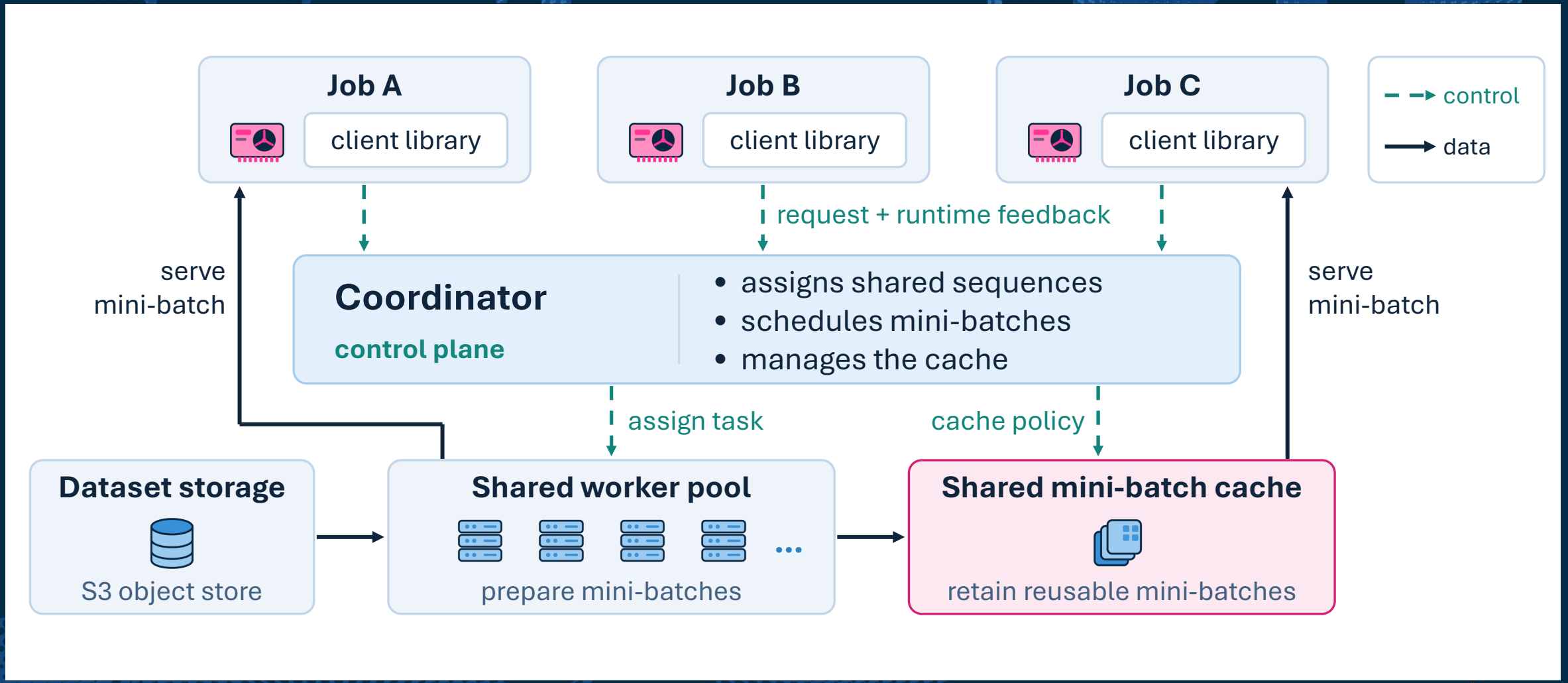


Benefit-aware caching

Prefetch and cache mini-batches by expected gain

Goal: maximize aggregate training throughput across jobs

BatchFlow Architecture



Centralized Sampling Makes Future Requests Known

The coordinator owns sampling and tracks each job's position

Coordinator: shared sequence



Prefetch ahead
prepare mini-batches before they are requested

Keep what's needed
know who needs each cached mini-batch next

Prepare each once
requests join work already in progress

Central sampling turns future requests into scheduling information

Benefit-Aware Worker Allocation

1 Estimate rates

$r_j,\ c_j$

job demand vs. worker supply

2 Set target lookahead

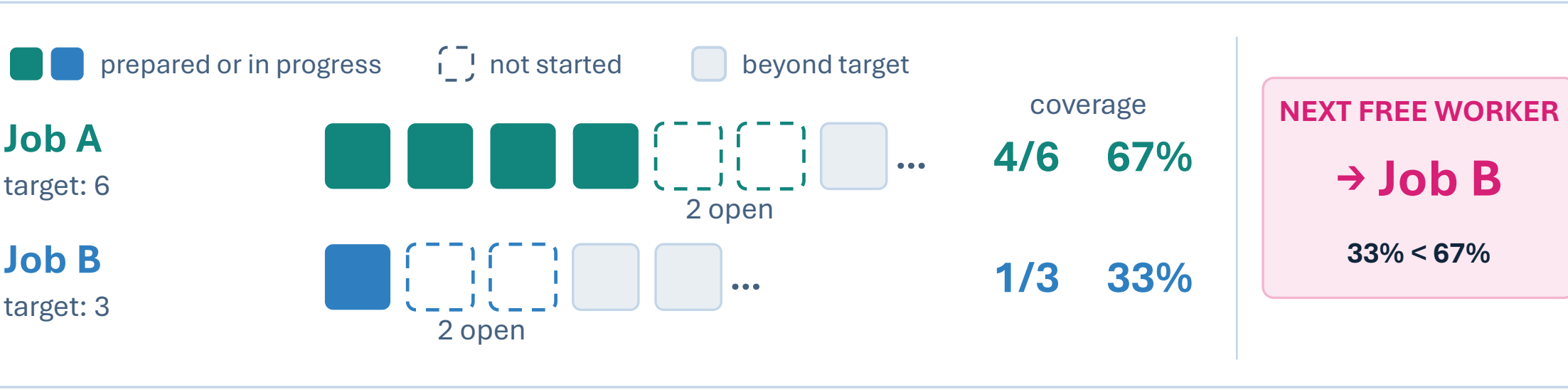
$d_j = \max(1,\ \lceil r_j/c_j \rceil)$

upcoming mini-batches to cover

3 Prioritize lowest coverage

$\text{minimize}\ \ p_j / d_j$

p_j = prepared or in progress



Allocate the next worker to the job with the lowest target coverage

Benefit-Aware Prefetching and Caching

Batch value

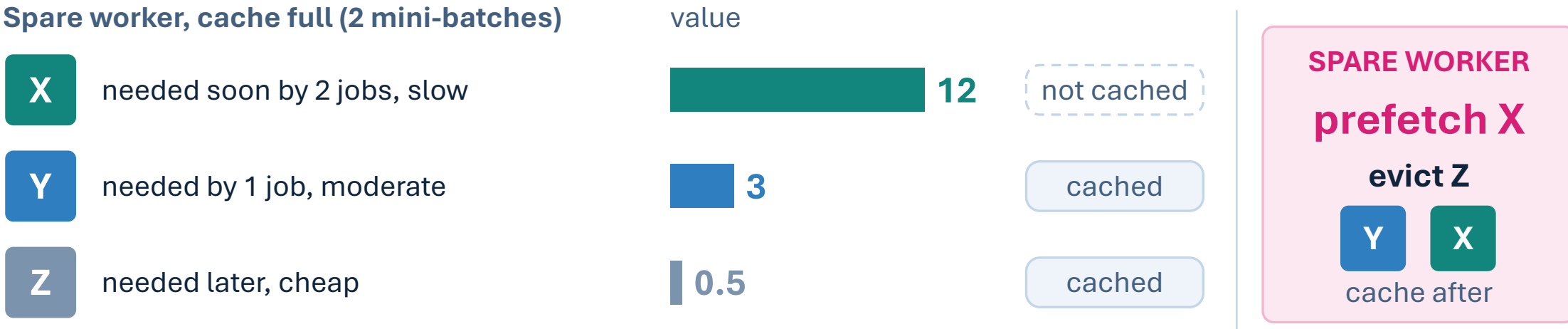
$$V(b) = \hat{h}(b) \cdot \hat{t}(b) \cdot (1 + \beta\, \hat{s}(b))$$

$\hat{h}$ **future demand** sooner, by more jobs

$\hat{t}$ **preparation time** slow to prepare

$\hat{s}$ **coverage urgency** jobs falling behind

Spare worker, cache full (2 mini-batches)



One value score decides what to prefetch and what to keep

Using BatchFlow in PyTorch Takes a Few Lines

Minimal client changes; the training loop remains familiar

1 Connect to BatchFlow

```
bf = BatchFlow(dataset_id=...)
```

2 Use the standard DataLoader

```
dl = DataLoader(bf.dataset())
```

3 Train normally + report metrics

```
for batch in dl:
    metrics = train_step(batch)
    bf.update_metrics(metrics)
```

Implementation details



gRPC

clients fetch prepared mini-batches from workers



Stateless workers

keep no job state; failed tasks are rerun



Redis cache

shared prepared mini-batches on Amazon ElastiCache

Training-step feedback informs worker scheduling and caching decisions

Evaluation Setup

AWS testbed



Training node
p3.8xlarge · 4× V100



Worker node
c5.12xlarge · 48 vCPUs



Shared cache
ElastiCache Redis



Dataset storage
Amazon S3

Workloads 4 concurrent jobs each

W1 **ImageNet-1K**
ResNet-18/50, MobileNetV3, VGG-16

W2 **Open Images**
ViT-B/32, ViT-S, LeViT, Mixer-B/32

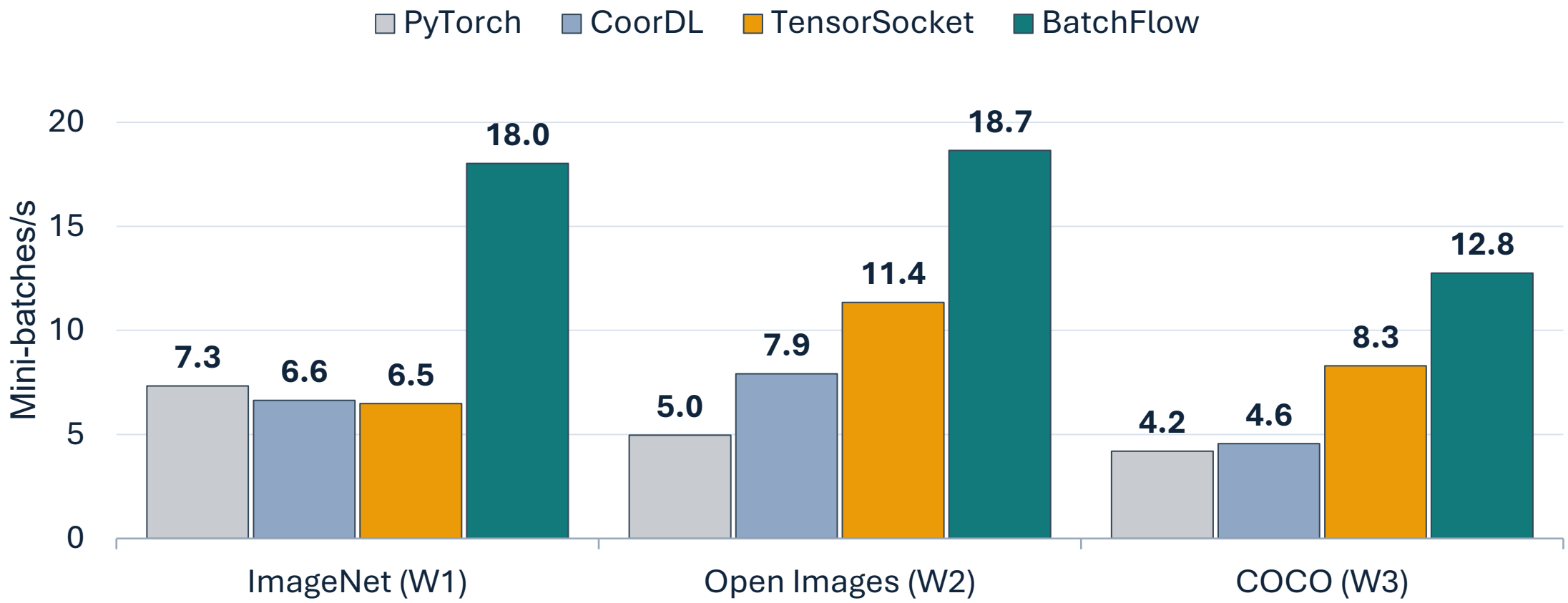
W3 **COCO**
ALBEF with 2/4/8/16-layer encoders

Baselines PyTorch DataLoader (per-job) · CoorDL (VLDB ’21) · TensorSocket (SIGMOD ’25)

Metrics aggregate throughput (mini-batches/s) · cost efficiency (mini-batches/$)

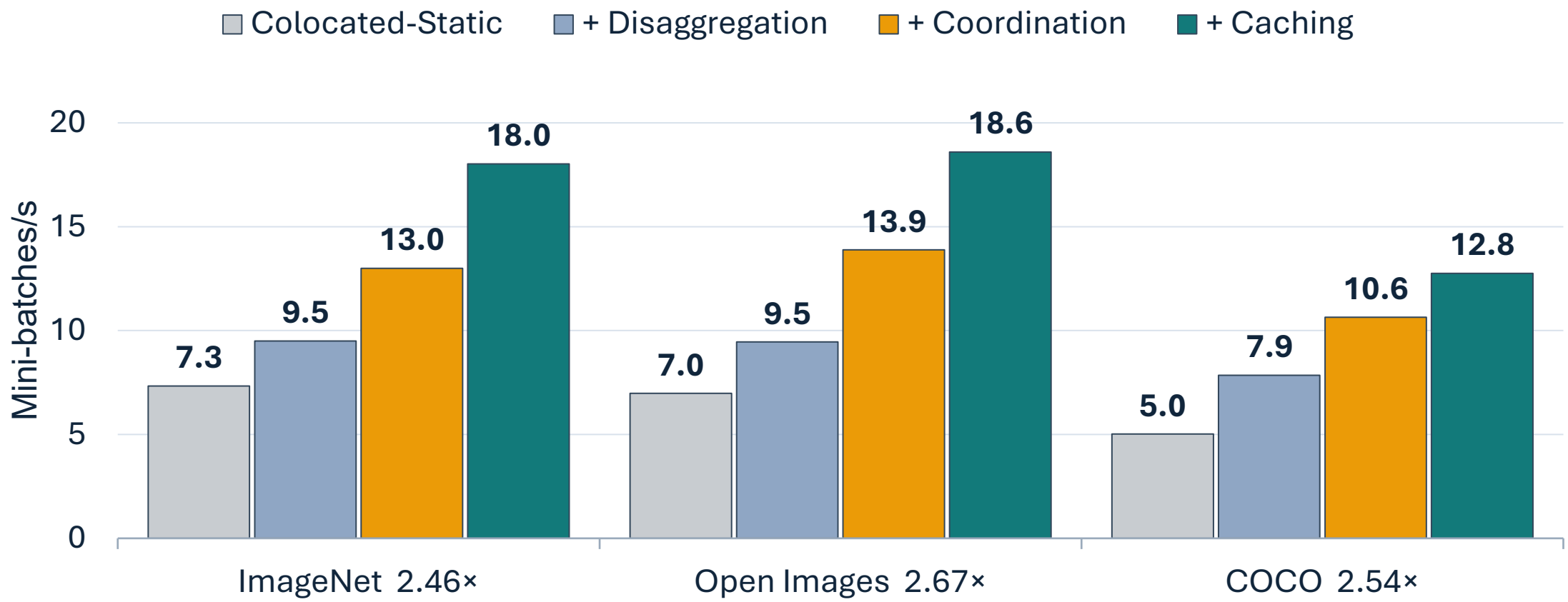
BatchFlow Leads on All Three Workloads

Up to **3.8×** higher aggregate throughput and **2.1×** higher cost efficiency



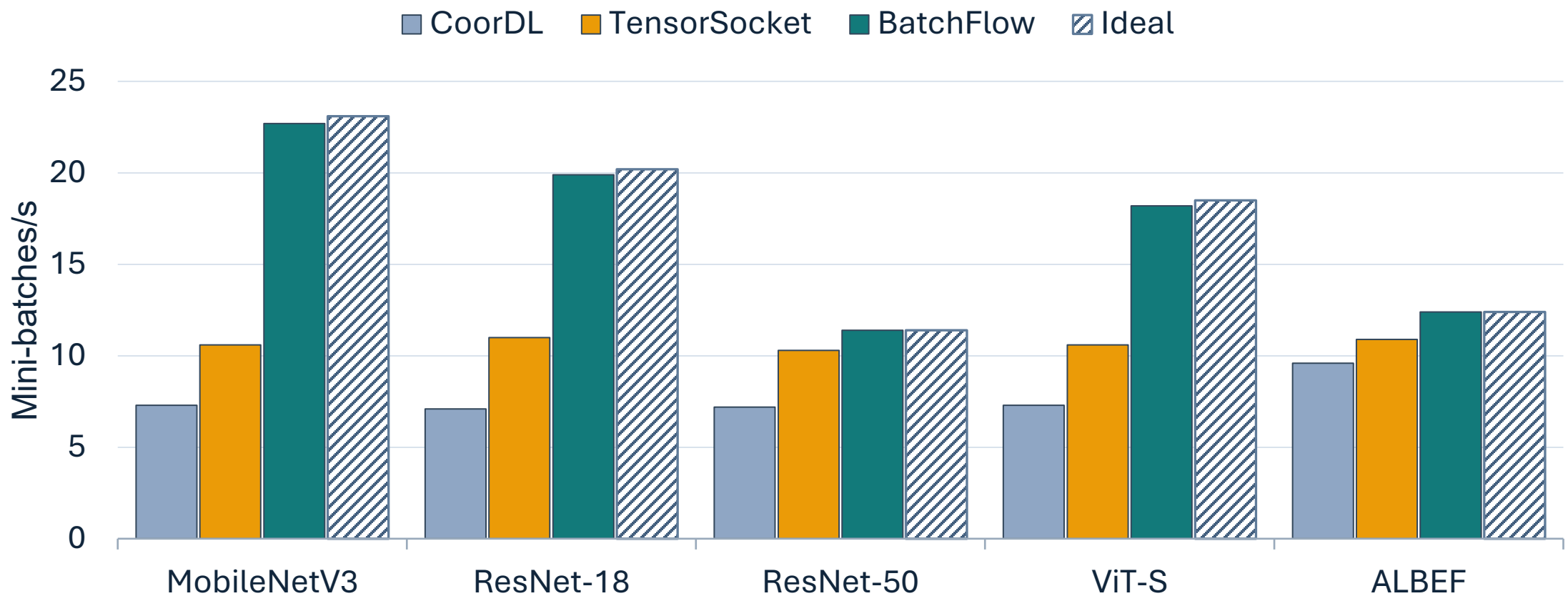
Every Mechanism Contributes

Each step adds one mechanism; all but the first use 48 workers



Even in the Best Case for Reuse, BatchFlow Leads

1.1–2.1× vs TensorSocket · **1.3–3.1×** vs CoorDL · within **2%** of Ideal

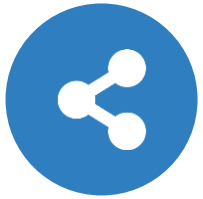


Future Work



Larger-scale evaluation

more jobs, GPUs and worker nodes



Distributed cache

spread the shared cache across nodes



Fairness and per-job targets

objectives beyond aggregate throughput

Conclusions

Coordinate data preparation across jobs

Prioritize resources where they improve throughput most



Shared worker pool



Benefit-aware allocation



Benefit-aware caching

up to **3.8×**

higher aggregate throughput

up to **2.1×**

higher cost efficiency

BATCHFLOW

Thank You

Patrick Watters | pwatters@unr.edu

Feng Yan | fyan5@central.uh.edu

Supported by the U.S. National Science Foundation: CAREER-2305491, CNS-2504762, OIA-2148788, CCF-2502257

University of Nevada, Reno

IBM

NSF EPSCoR
ADVANCING GEOGRAPHIC DIVERSITY IN STEM

SC26 hpc unites | Chicago