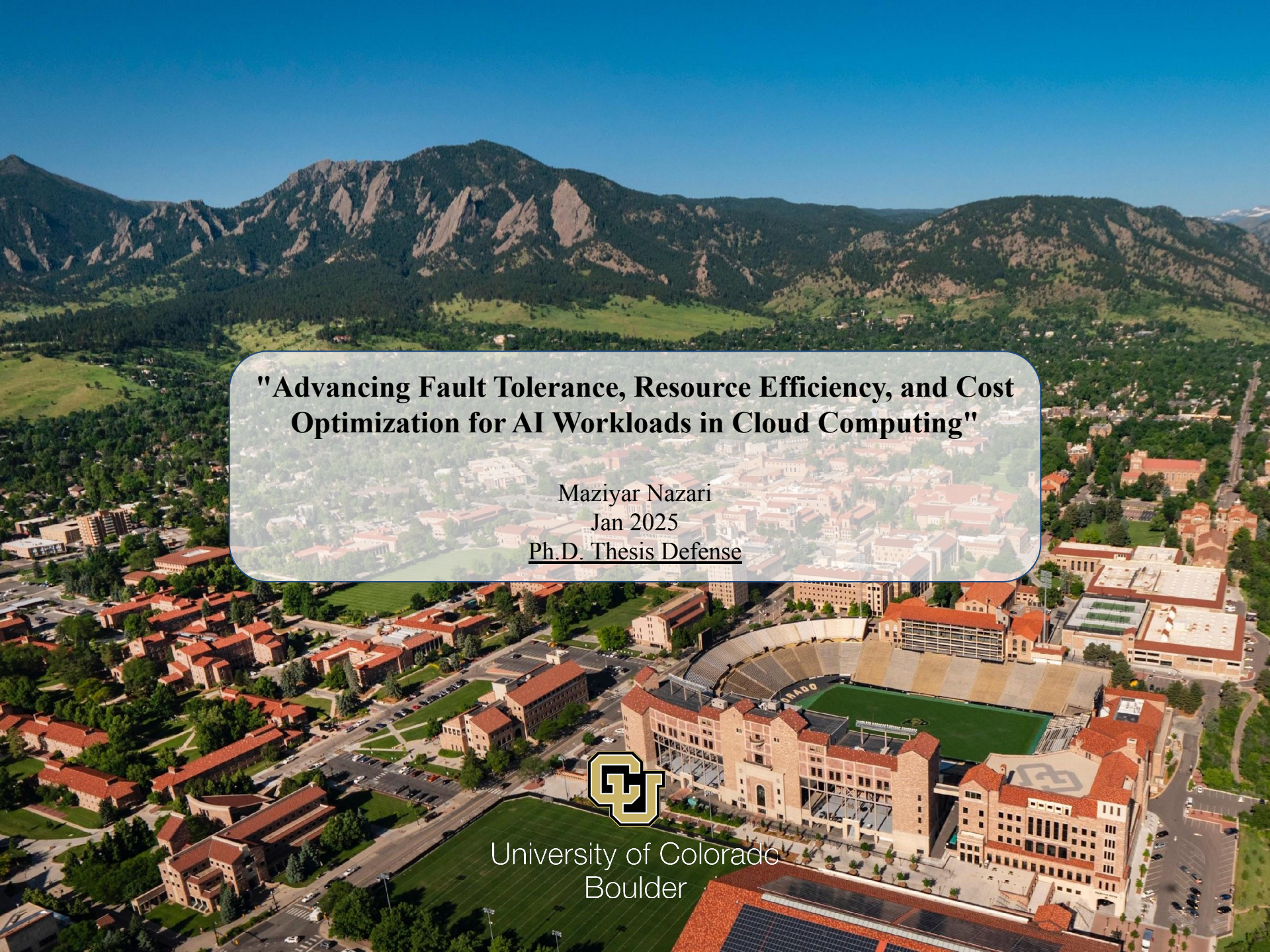


"Advancing Fault Tolerance, Resource Efficiency, and Cost Optimization for AI Workloads in Cloud Computing"

Maziyar Nazari

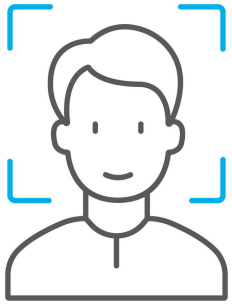
Jan 2025

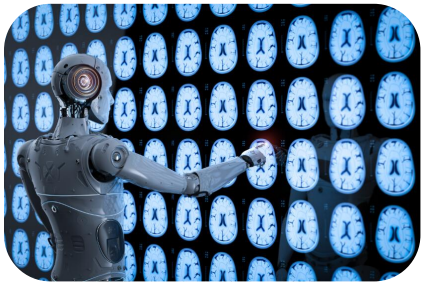
Ph.D. Thesis Defense

University of Colorado
Boulder

Introduction

Deep Learning Applications are Everywhere!





Introduction

2 Parties in Machine Learning Application Development Lifecycle

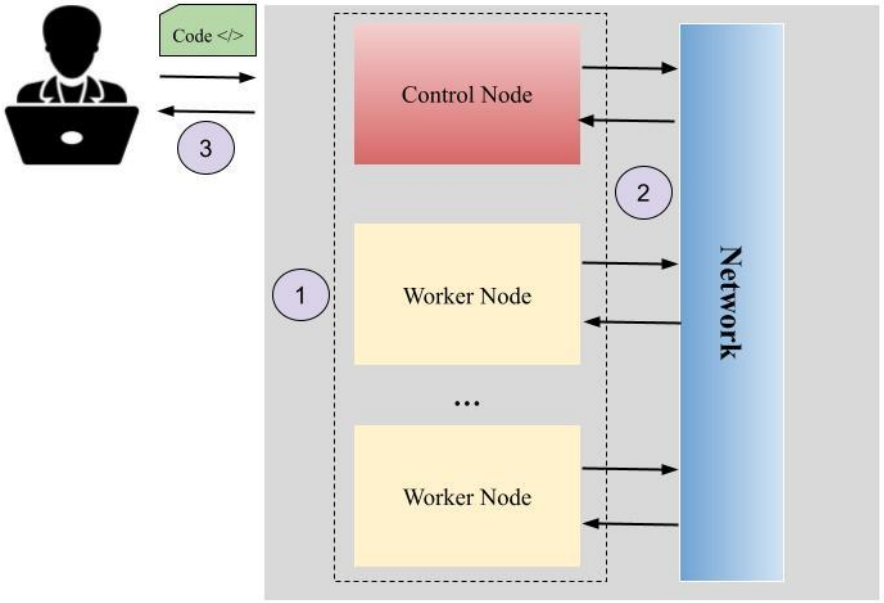
Developer

- **Data Collection**
- **Data Preprocessing**
- **Model Development and Deployment**
- **Pick the right cloud configuration**
- **Monitoring**
- **Hyperparameter Tuning**

Cloud Provider

- **Resource Management**
- **Service Level Agreement**
- **Cost Effective**
- **User-friendly Interface**
- **Rich API**

Overview



1. Resource allocation is suboptimal when running (Distributed) ML/DL workload on containerized infrastructure
2. SOTA network fault tolerance mechanism in (Distributed) ML/DL workloads is inefficient
3. Picking the right cloud configuration is complicated!

Introduction

Challenge	Solution
Resource allocation is suboptimal when running (Distributed) ML/DL workload on containerized infrastructure	Event-driven Sub-second Container Resource Allocation
SOTA network fault tolerance mechanism in (Distributed) ML/DL workloads is inefficient	Transparent Hardware Offloaded Resilient Networks for RDMA based Distributed ML Workloads
Picking the right cloud configuration is complicated!	Optimizing Cloud Configuration Selection for Cost-Effective Distributed Deep Learning Execution

Escra: Event-driven Sub-second Container Resource Allocation

[Published in ICDCS 2022]

Escra: Event-driven, Sub-second Container Resource Allocation

Greg Cusack, Maziyar Nazari, Sepideh Goodarzy, Erika Hunhoff,
Prerit Oberai, Eric Keller, Eric Rozner, Richard Han
University of Colorado Boulder, Boulder, CO USA

Abstract—**This paper pushes the limits of automated resource allocation in container environments. Recent works set container CPU and memory limits by automatically scaling containers based on past resource usage. However, these systems are heavy weight and run on coarse-grained time scales, resulting in poor performance when predictions are incorrect. We propose Escra, a container orchestrator that enables fine-grained, event-based resource allocation for a single container and distributed application. Escra can easily adapt to sub-second intervals and resource allocation can cost-effectively scale resources within and across host, meaning operators can cost-effectively scale resources without performance penalty. Escra integrates into and manages two types of containerized applications – microservices and serverless functions. In microservice environments, our evaluations show fine-grained and event-based resource allocation can reduce application latency by up to 96.9% and increase throughput by up to 3.2x when compared against the current state-of-the-art. Escra can increase performance while simultaneously reducing 50th and 99th%ile CPU waste by over 10x and 3.2x, respectively. In serverless environments, according to our evaluation, Escra can reduce an application's CPU reservations by over 2.1x and memory reservations by more than 2x while maintaining similar end-to-end performance.**

I. Introduction

Containerized infrastructure is quickly becoming a preferred method of deploying applications. The light-weight nature of containers coupled with rich orchestration systems enable a new way to design automated operations that are integrated with development workflows. In these deployments, per-container resources limits are used to prevent interference between containers and unchecked resource usage.

Setting container resource limits is a trade-off between application performance and efficient use of underlying system resources. When resource limits are set low to prioritize efficient resource use, applications will experience an increased number of CPU throttles and out-of-memory (OOM) events. Throttles slow processing and $OOMs$ kill containers which results in degraded application performance. When resource limits are set high to prioritize application performance, resources are underutilized which increases deployment cost. Developers pay the cost when cloud providers charge tenants based on resources reserved [1]–[3]. Cloud providers pay the cost in cases where developers are charged by usage, such as in serverless computing [4]–[7].

Due to this trade-off, setting accurate limits is important. In practice, it is also difficult [1], [8]–[11][1]. Using profiling to characterize application resource requirements will only result in accurate estimates if there is a representative workload. As workloads are often dynamic, the resources needed will change over long timescales (diurnal patterns, gradual changes in application popularity, etc.) and short timescales (bursts, failures of coupled systems, etc.). Since creating an accurate estimate of resource requirements is so complex, developers and operators often resort to over provisioning resources. This results in underutilized deployments which are often observed by datacenter operators [10], [12]–[14].

Recent work has addressed some of these challenges [1], [9] by leveraging machine learning to predict future needs and then automatically scaling container resource limits based on those predictions. These works eliminate the developer burden of setting resource limits but are constrained to using coarse-grained intervals (e.g., several minutes) to set resource limits. Coarse-grained intervals are required because the system has to learn enough information to be able to predict resource use. This is a poor fit for some workloads with short-lived containers, such as in serverless systems [15]–[18]. Coarse-grained intervals also increase the odds of misprediction since the dynamics of applications can change throughout an interval. Thus, these works still contend with the performance and efficiency trade-off.

In this paper, we argue the performance and efficiency trade-off can be avoided by using a *fine-grained, event-based resource allocation* scheme. To this end, we introduce *Escra*: a fine-grained, event-based resource allocation infrastructure for single containers and distributed resource allocation capable of managing containers' resources across multiple nodes. We find resource allocation can easily adapt to sub-second intervals within and across hosts, allowing datacenter operators to cost-effectively scale and assign resources without performance penalty. This scheme has numerous benefits. Instead of a container being killed when it reaches an OOM event, an *event-based* system can catch the event and scale the container dynamically. Instead of making conservative allocations in order to avoid performance degradation over coarse-grained time intervals, a *fine-grained* system can always aim to tightfit allocations to current resource demands, and can quickly react to instances of CPU throttling.

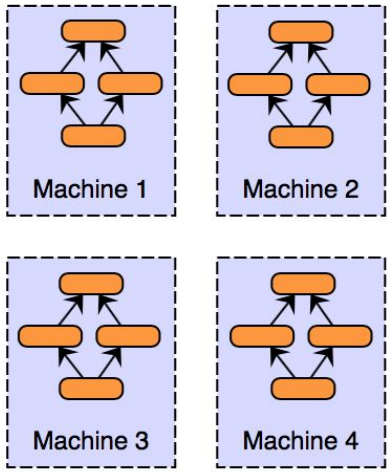
Escra's design consists of a logically centralized controller that administers resource allocations to containers across servers. Each server is instrumented with kernel hooks and runs an agent process to apply resource decisions and report container usage to the controller. A *Distributed Container* abstraction enforces per-application resource isolation and allows

[1]The aggregate CPU utilization at Twitter is <20% but the reservations reach up to 80%. Memory utilization is only slightly better at 40-50% but the reservations still greatly exceed the usage [10].

Distributed Deep Learning

Background

Data Parallelism

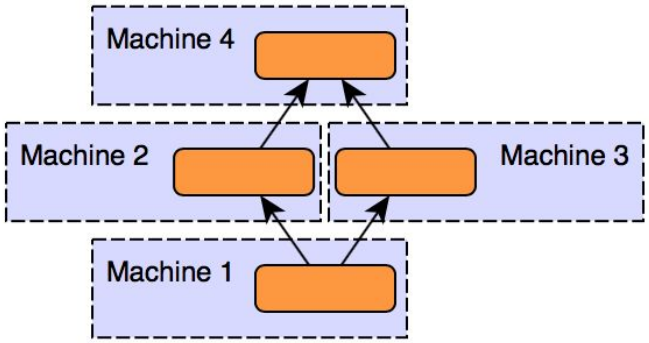


E.g.

PyTorch

TensorFlow

Model Parallelism



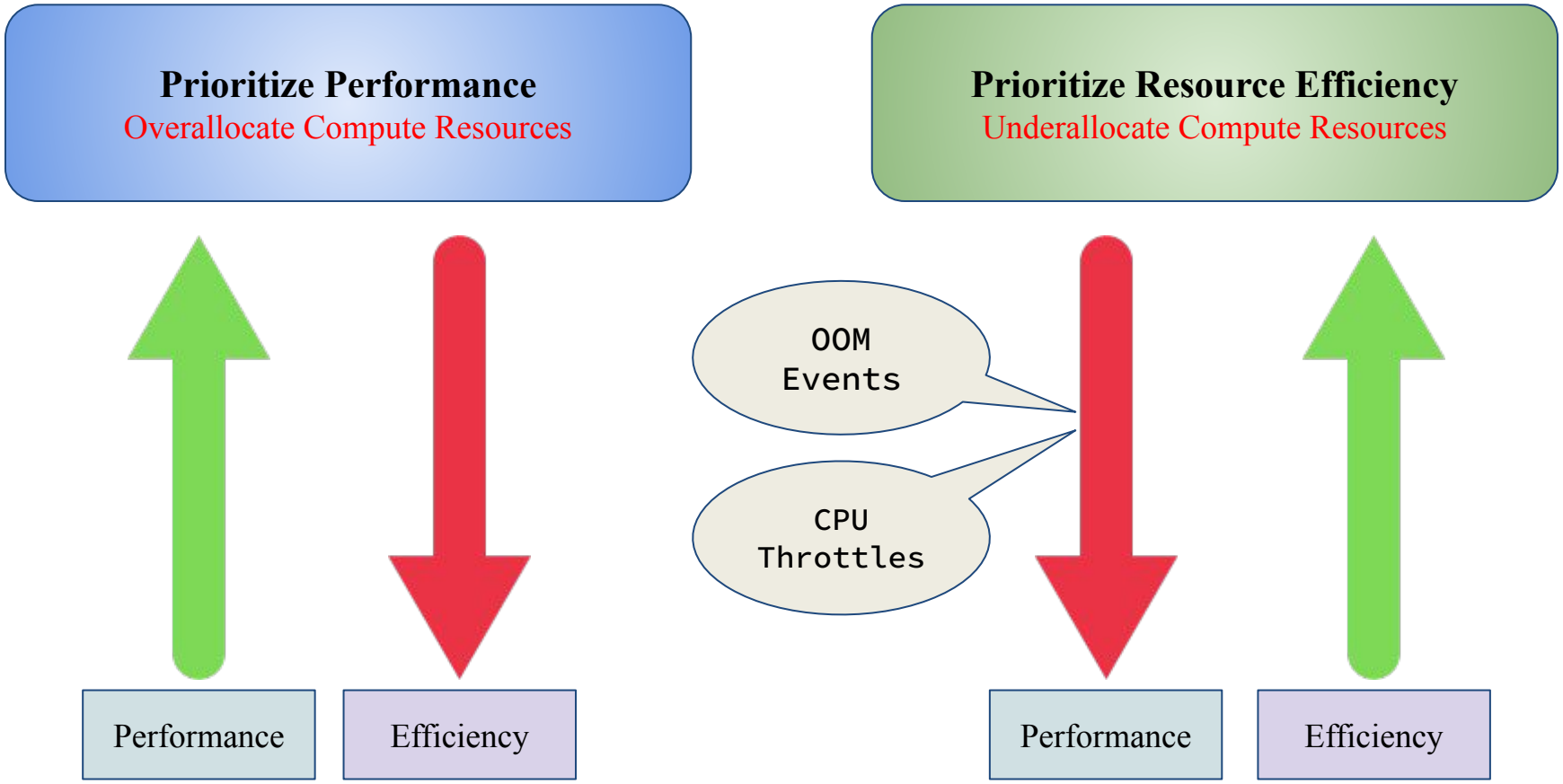
Background

Containerized Infrastructure

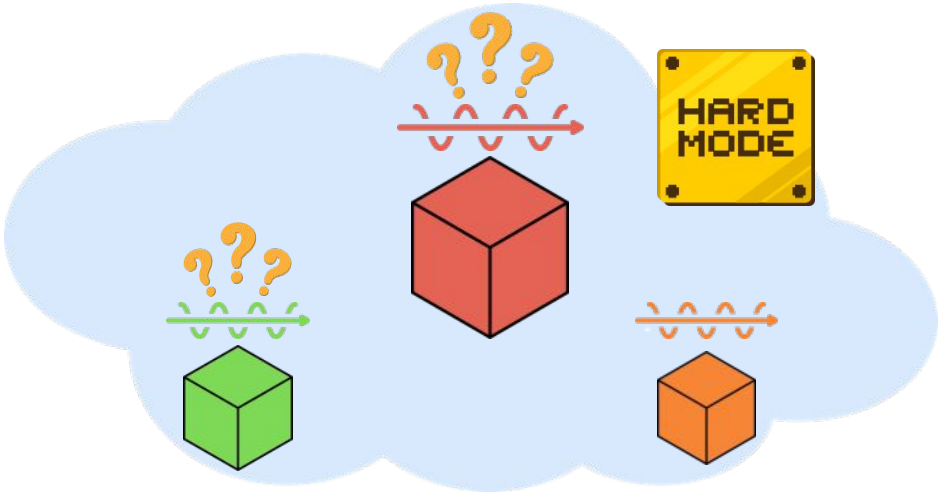
- Light-weight
- Rich Orchestration Systems
- Development Workflow Integration
- Strong Resource Isolation

Introduction: Problem Statement

Developers must make trade-offs!



Introduction: Problem Statement



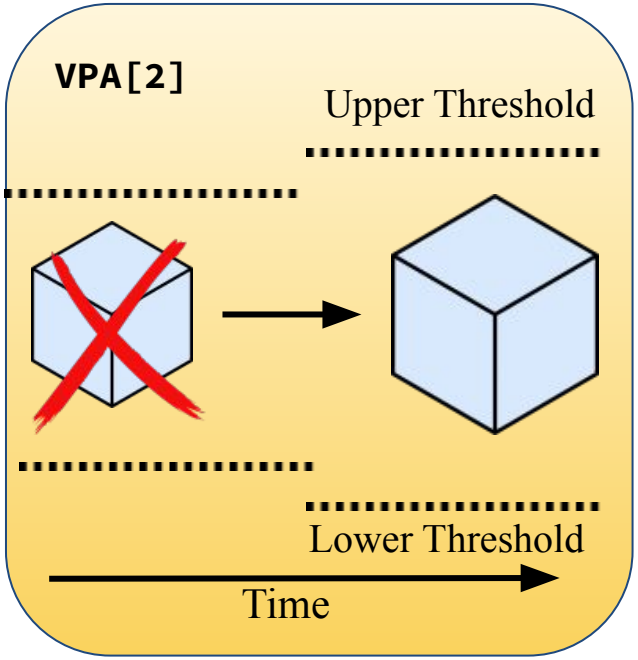
Setting Container Limits is Hard!

- Requires representative workload
- Resource needs vary over time
- Tools aggregate usage over coarse-grained timescales

→ Overprovisioned Containers

Related Works

Vertical Pod Autoscaler!

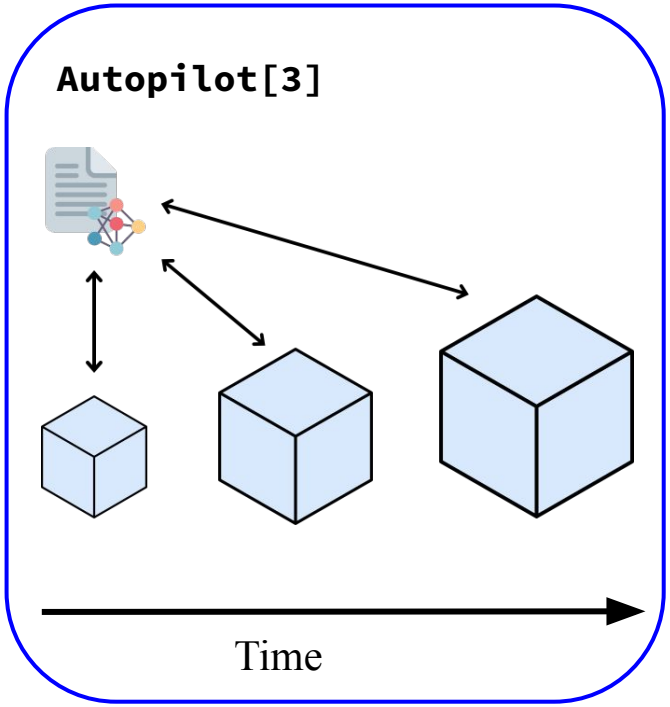


Shortcomings:

- Containers restart to scale
- Scale once per minute
- High slack

Related Works

Google Autopilot: RL-like Algorithm



Shortcomings:

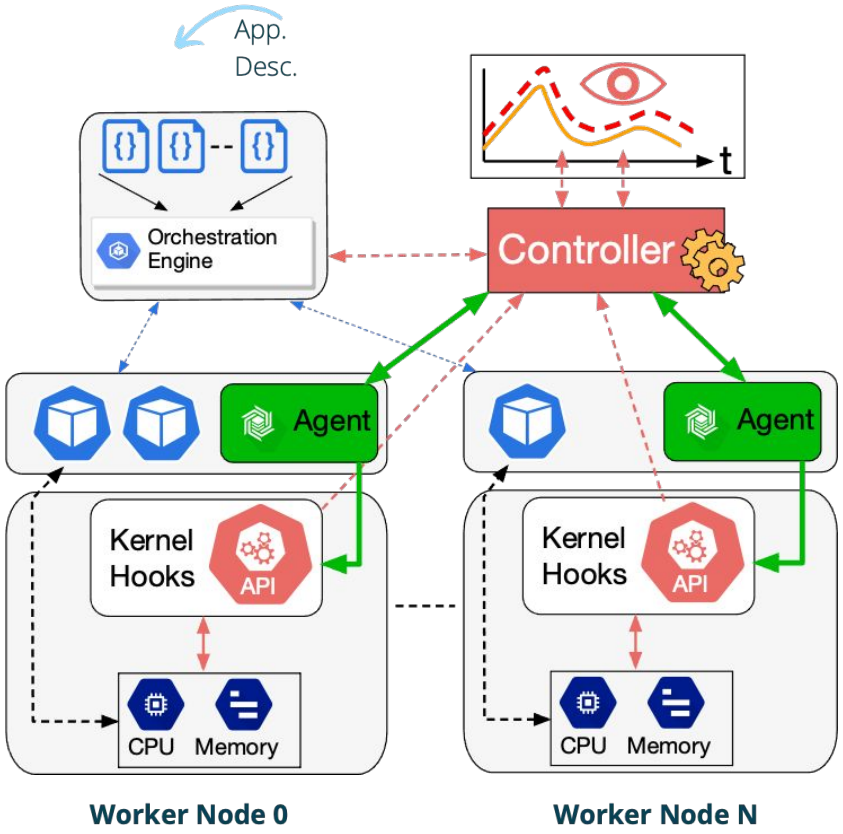
- Cannot respond to quick changes in the workload
- Must allocate to maximum prediction in the next 5-min interval
- Unable to correct inaccurate predictions
- Not suitable for short-lived containers

Event-driven Sub-second Container Resource Allocation

Escra:

- Distributed Resource Allocation
- Sub-second interval resource scaling within and across hosts
- OOM prevention and scaling
- Immediate CPU throttle response
- No performance penalty

System Design



Escra Architecture:

- Kernel hooks
- Controller
- Resource allocator
- Agent

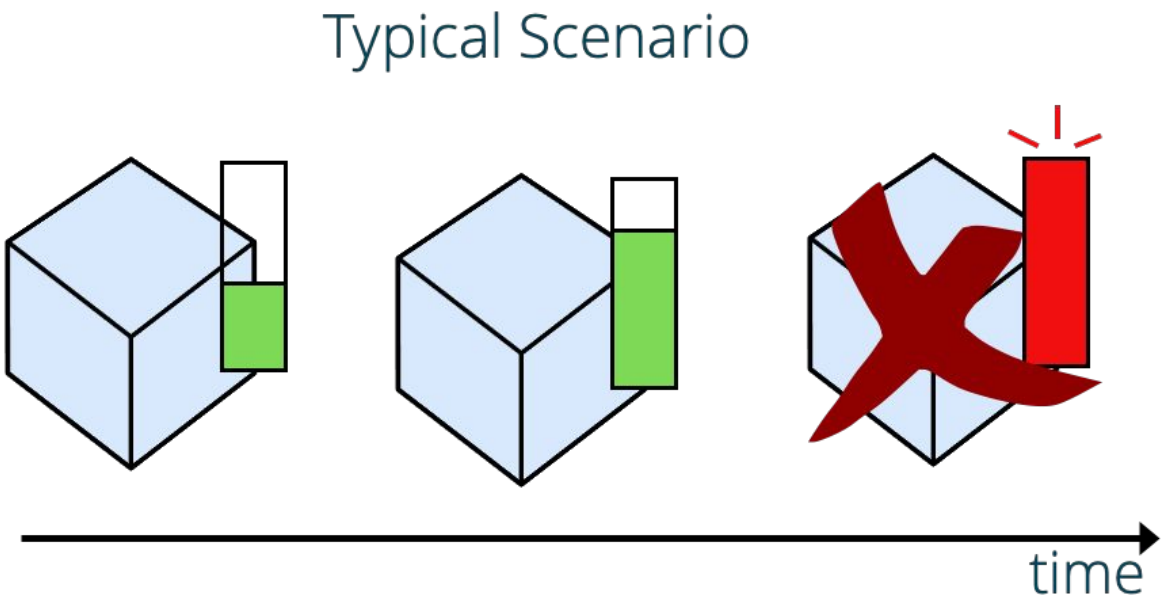
Dynamic Resource Allocation Process

Dynamic CPU Allocation:

- Scale up and down the CPU quota
- Based on sliding window algorithm over n previous periods
  - Fine-grained, per period interval telemetry data
    - *Past quotas*
    - *If the container is throttled*
  - Refer to the paper for the formula details
- Respects the total cpu limit in our distributed setting

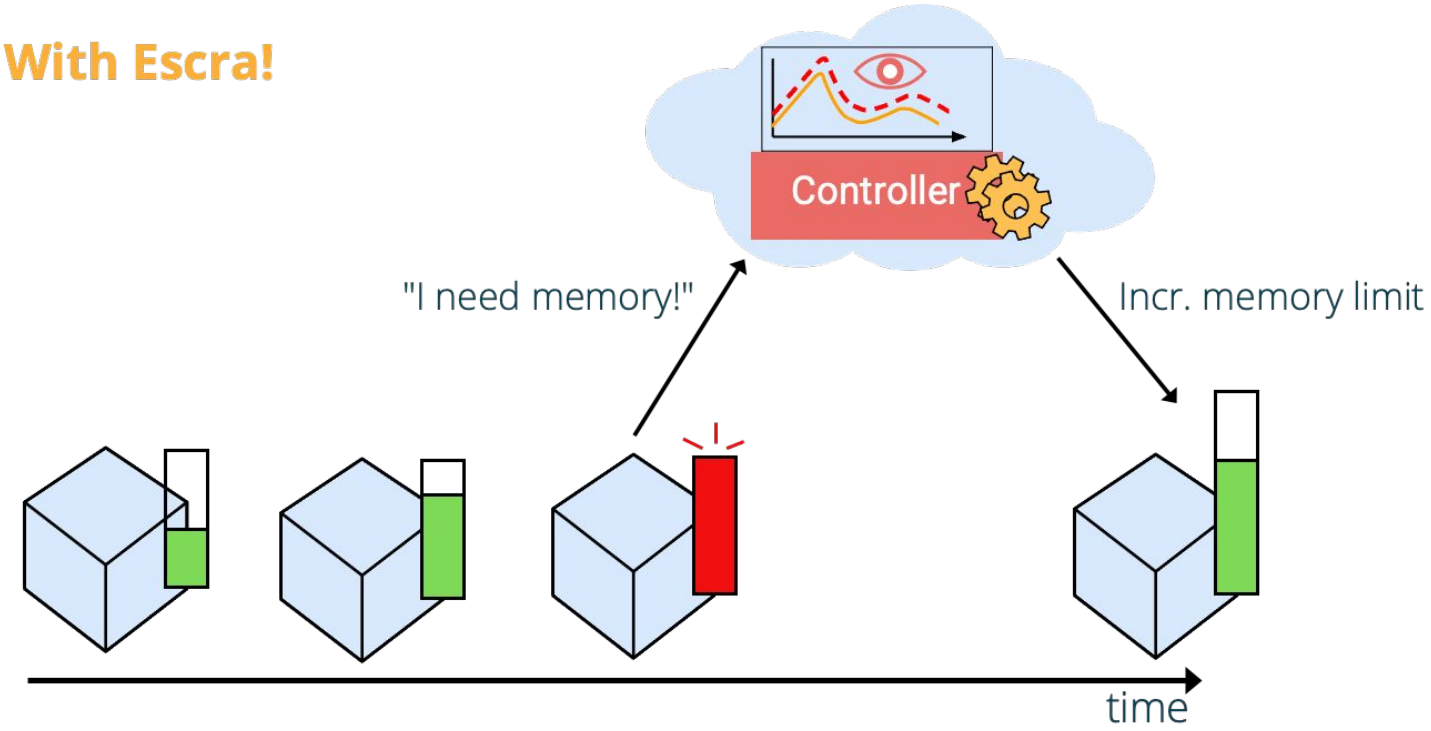
Dynamic Resource Allocation Process

Event-based Memory Allocation:



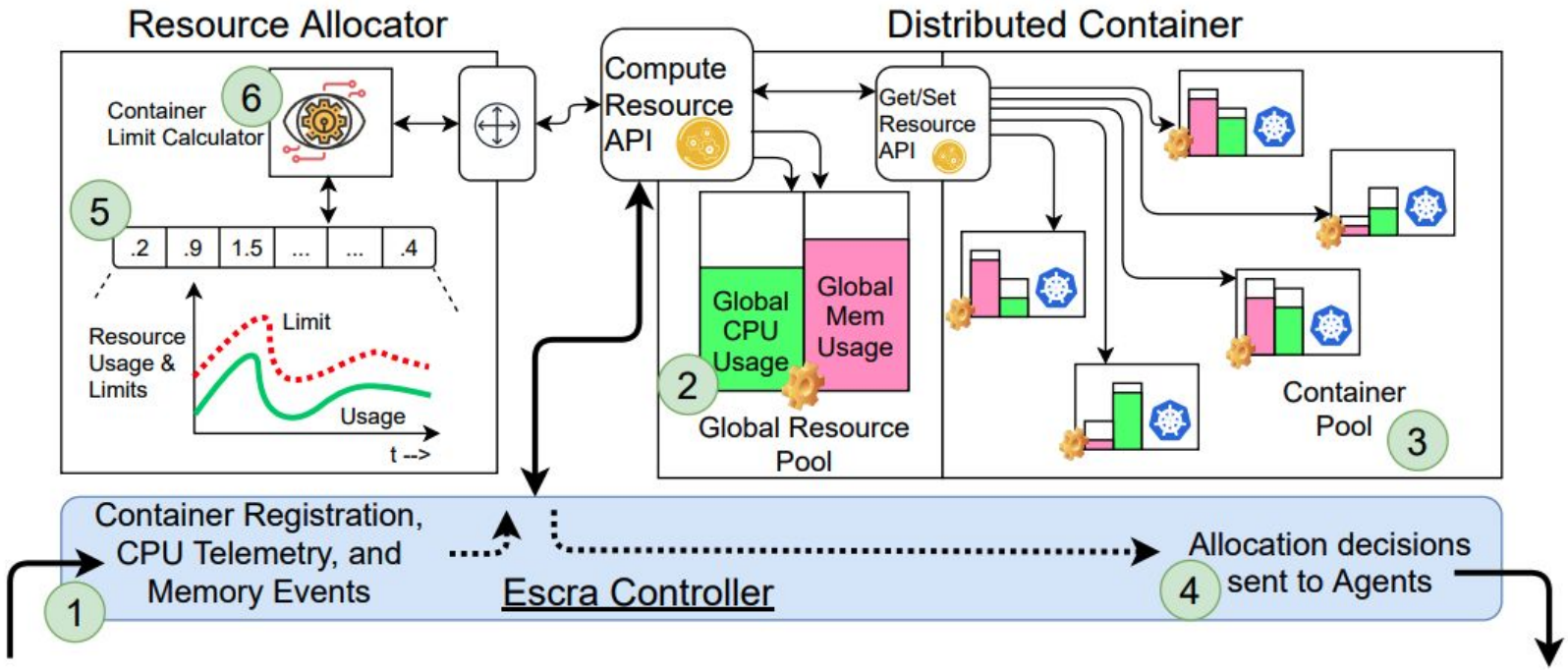
Dynamic Resource Allocation Process

Event-based Memory Allocation:



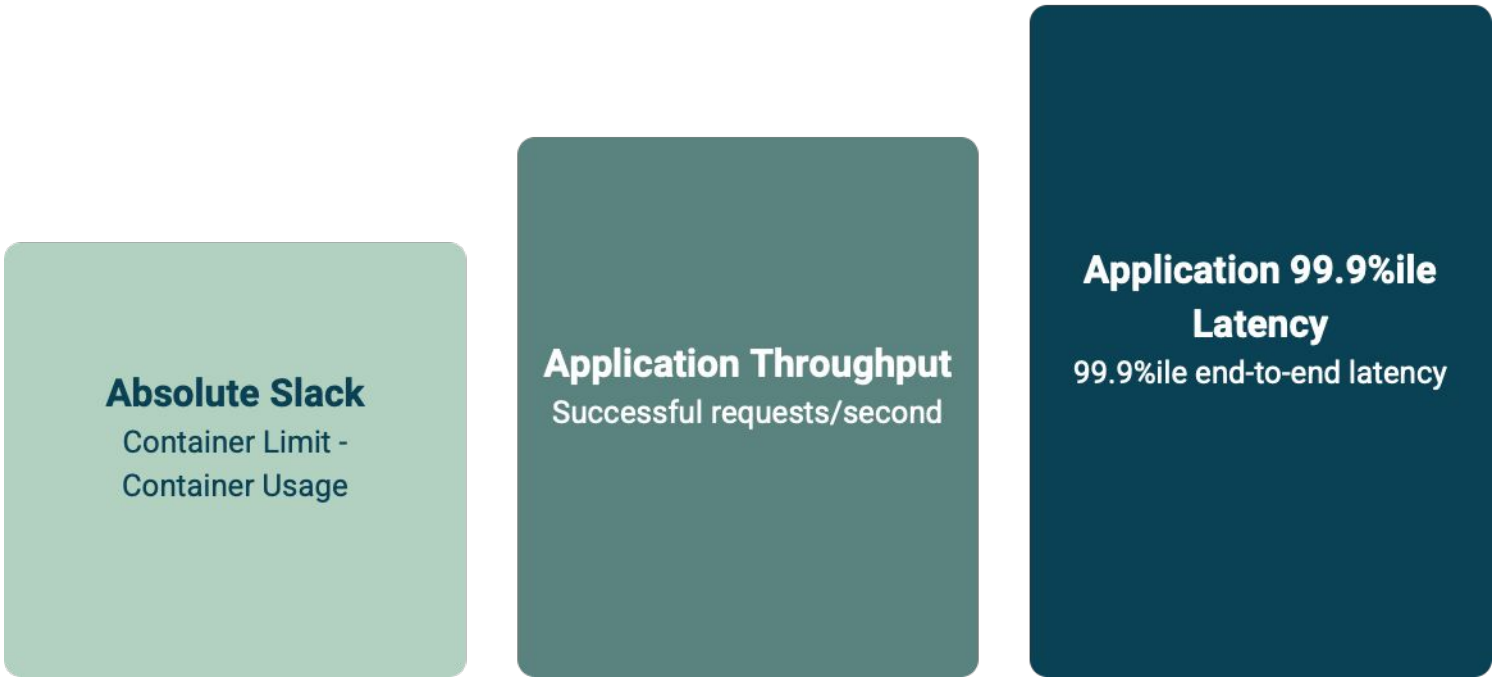
Escra

Escra Workflow and Components:



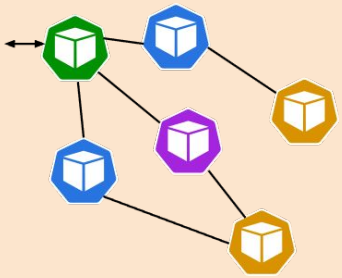
Escra Evaluation

Benchmarks Metrics

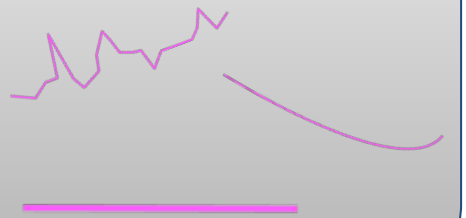


Applications and Workloads

Applications:
- TrainTicket
- TeaStore
- HipsterShop
- MediaMicroservice



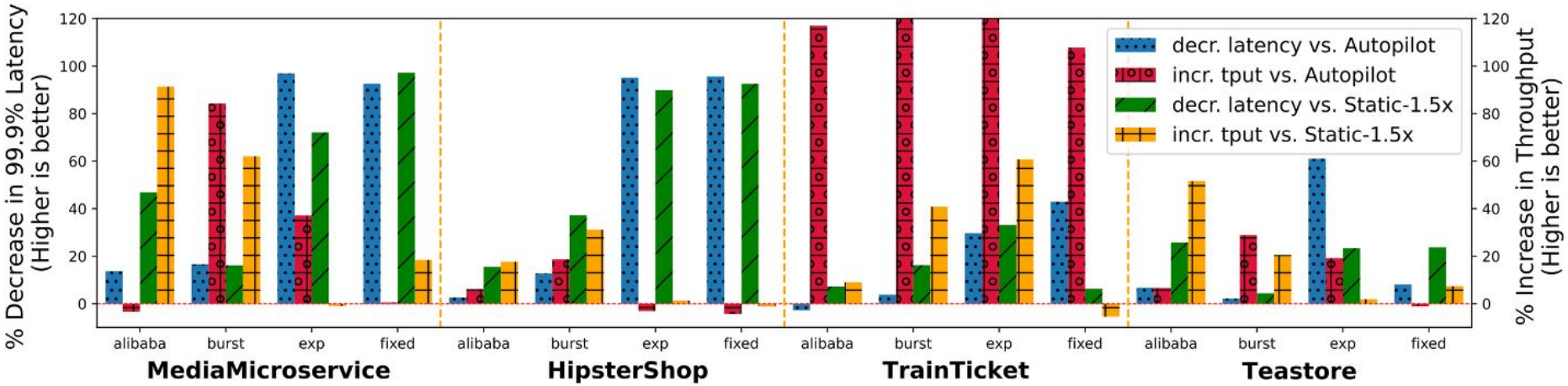
Workloads:
- Alibaba
- Burst
- Fixed
- Exponential



Note: Escra Integrated with Kubernetes to run these apps

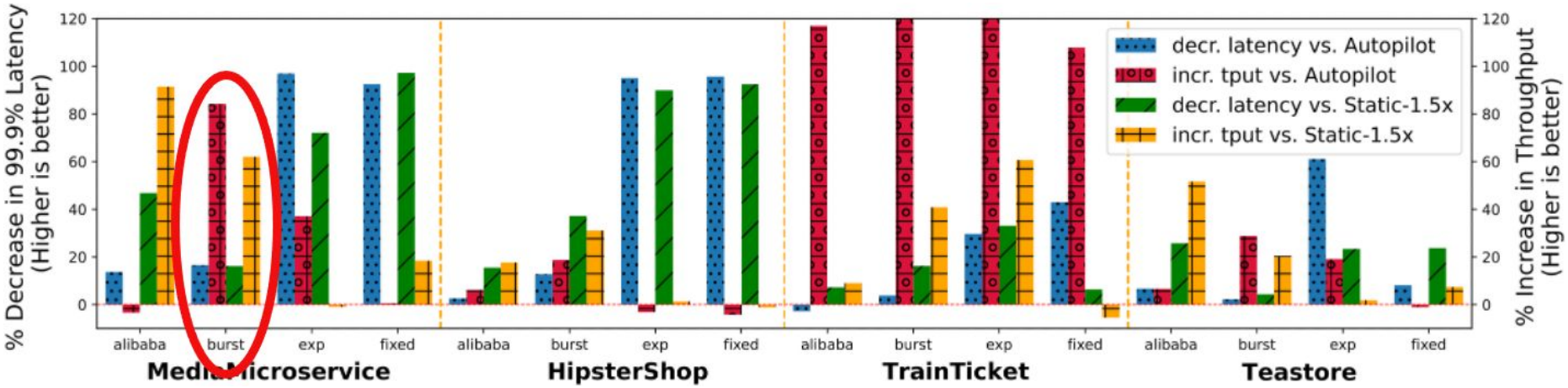
Application Performance Evaluation

Application latency/throughput comparison against Autopilot and static resource allocation



Escra Wins!

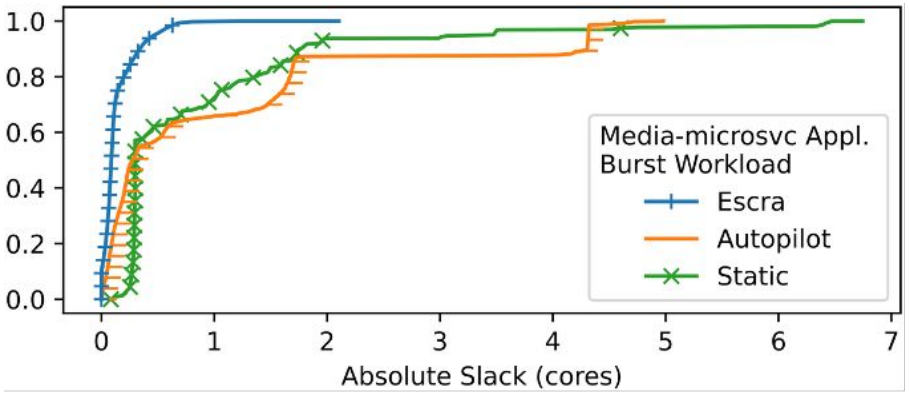
Microservice Benchmark: Burst Workload

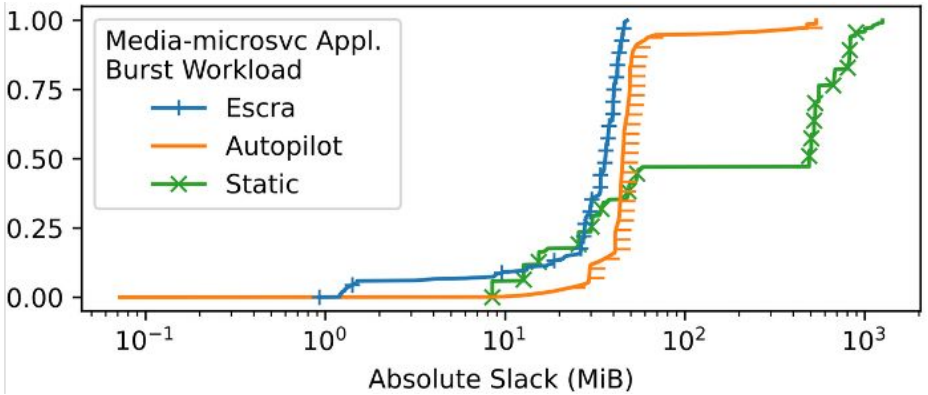


Escra Reduces Waste

Microservice Benchmark: Burst Workload

Escra vs Autopilot

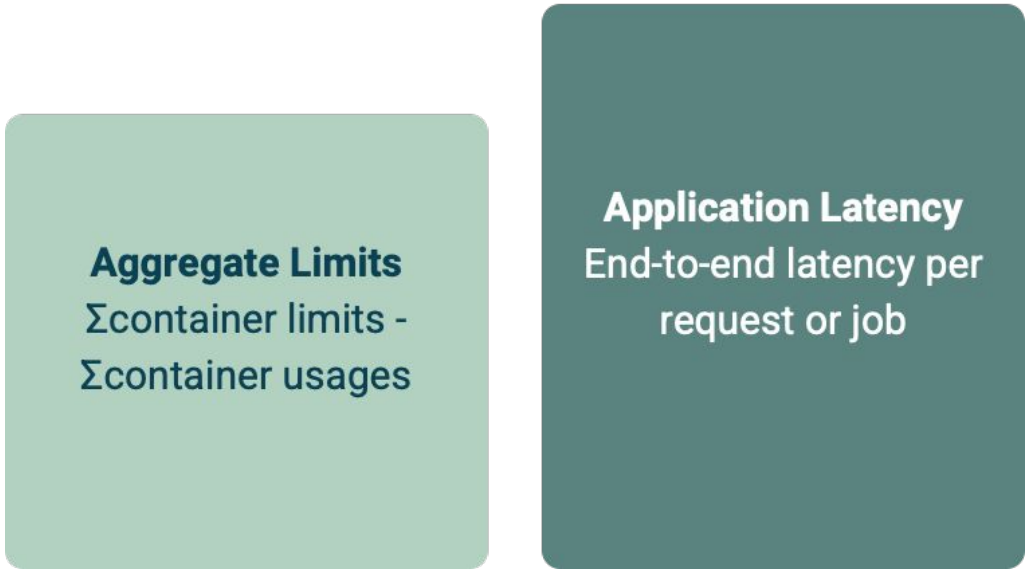




Application performance with Escra vs Autopilot (SOTA):
Throughput increase: 16.6%
Latency Decrease: 84.3%

Escra Evaluation

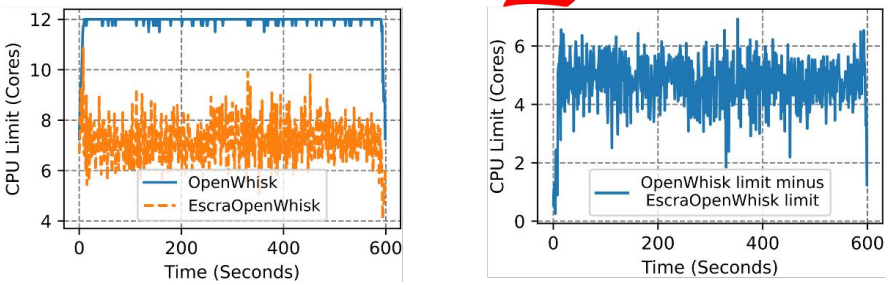
Serverless Benchmark Metrics



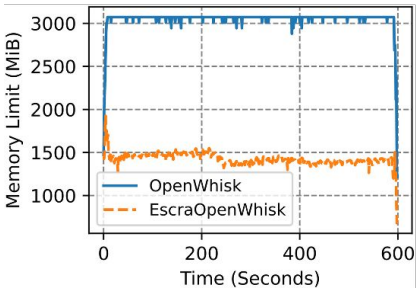
Escra Integration with OpenWhisk

Serverless Benchmark

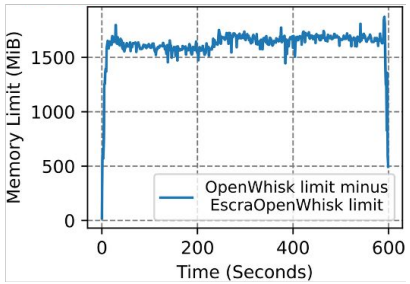
- **ImageProcessing Application**

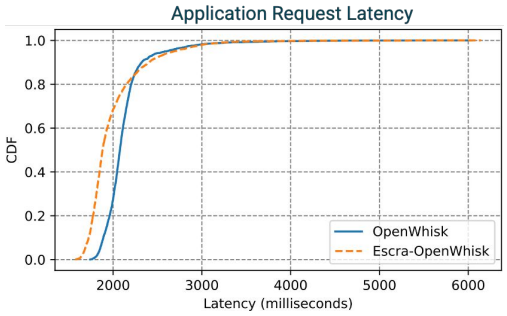


Difference!



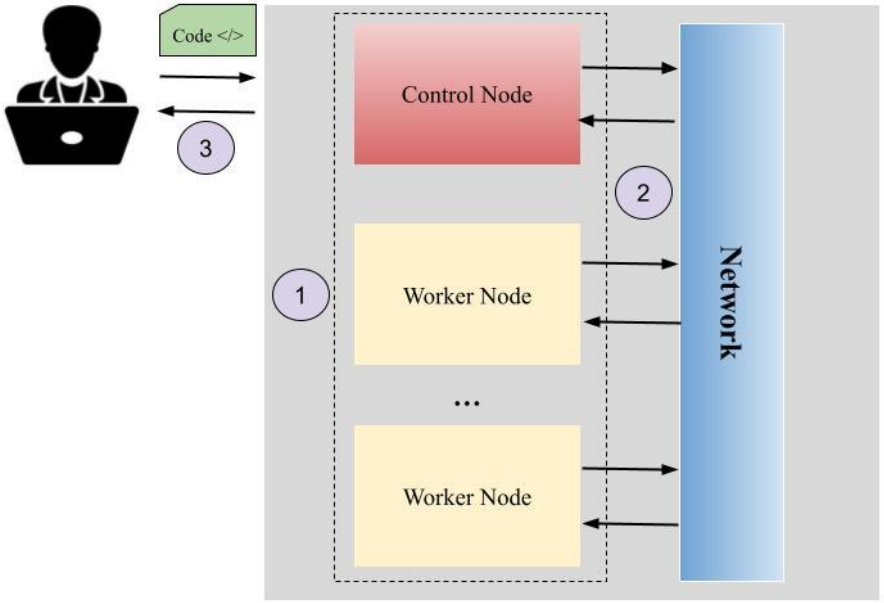
Difference!





- 80% of the times Escra + OW has strong performance gain
- Avg CPU Saving: 5 Cores
- Avf Memory Saving: 1550 MiB

Recap



1. Escra, Event-driven Sub-second Container Resource Allocation
2. THORN-ML: Transparent Hardware Offloaded Resilient Networks for RDMA based Distributed ML Workloads
3. Optimizing Cloud Configuration Selection for Cost-Effective Distributed Deep Learning Execution

THORN-ML: Transparent Hardware Offloaded Resilient Networks for RDMA based Distributed ML Workloads

[Under review @ MlSys 2025]

THORN-ML: TRANSPARENT HARDWARE OFFLOADED RESILIENT NETWORKS FOR RDMA BASED DISTRIBUTED ML WORKLOADS

Anonymous Authors[1]

ABSTRACT

Distributed deep learning (DDL) requires a great investment in infrastructure, including accelerated compute nodes and networking hardware capable of supporting high performance networking, e.g., Remote Direct Memory Access (RDMA). When a host running a DDL application becomes unreachable, the cost can be high as application-level failure recovery is slow and disruptive. When the host is unreachable due to host failure, this is unavoidable; however, when the network components involved in attaching the host to the core data center network fail, we argue that this cost is avoidable. This paper introduces THORN-ML, a hardware-offloaded resilient network architecture that is completely transparent to DDL applications and Mellanox ConnectX-5 NICs, with several THORN-ML on a cluster of 5 nodes with Nvidia A100 GPUs and works with commodity hardware. We evaluate applications leveraging model parallelism and/or data parallelism, and find that THORN-ML reduces disruption from minutes (impacting the whole cluster) to milliseconds (impacting packets that can be re-transmitted).

1 INTRODUCTION

Deep learning has seen tremendous growth, stemming from both advances in algorithms and advances in infrastructure – both of which support ever increasing sizes of models and training data. The clusters which run distributed deep learning (DDL) applications consist of racks of servers, each with network cards (NICs) connected to a top-of-rack (ToR) switch, which are then interconnected through a topology of core network switches. As a snapshot of the scale of this infrastructure, Meta recently invested in a cluster with 24,000 GPUs, with a goal of managing 350,000 total GPUs by the end of 2024 (Lee et al., 2024).

The large (and increasing) number of components increases the likelihood of failure of individual components. Given how failures are handled, this simultaneously increases the impact of a failure. Consider the case where a host fails. DDL frameworks such as TensorFlow (Tensorflow), Ray (Ray), and PyTorch (Pytorch), handle host-level failures by using distributed checkpoints. Training typically involves several iterations, or epochs, where forward and backward propagation occur in neural networks. A checkpoint of the state of the application (which is distributed across the cluster) is then recorded after each epoch or at specified intervals based on application configuration.

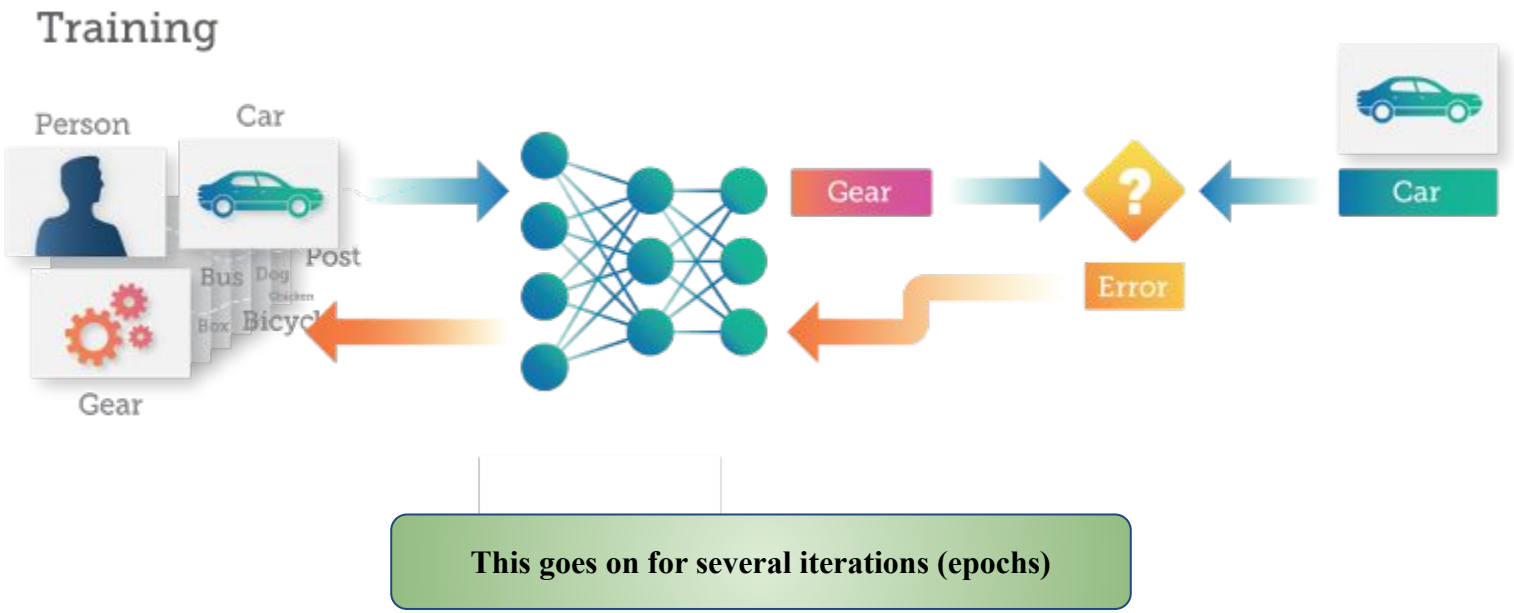
In the event of a failure, the DDL training job can restart from the last checkpoint, avoiding the need to start from scratch. However, this mechanism comes with significant overhead, as all workers must revert to the checkpoint. In our measurements on a 5 node GPU cluster training the GPT-2 model and ResNet101, a single epoch takes anywhere from 4 to 80 seconds, and the time to initialize all nodes running workers takes around 21 seconds (details in Section 2). Combined, this is the time the resources reserved for this job cannot perform any useful computation. Our measurements are likely underestimating these times as (1) these measurements assume taking a checkpoint every epoch, which has I/O, storage, and compute overhead, so in practice, checkpoints are likely to be performed at longer intervals, and (2) these measurements were gathered on a small cluster, and are likely to be higher on bigger clusters.

On the network side, core data center networks are designed with redundancy and routing protocols to detect and route around failures with little disruption (Gruenberg et al., 2009; Poutievski et al., 2022; Dutt, 2018). However, redundancy and protocols do not extend to the edge of the network, leaving the connectivity between the host and the ToR switch vulnerable to failures (accounting for nearly 10% of overall disruptions, based on measurements from Meta (Dubey et al., 2024).). The components at this connection point include the NIC on each host, the ToR switches themselves, and the cables that connect the NICs to the ToR switches. Current practice detects the hosts as unreachable, whether due to the host failure or failure at the network edge, and as such, treat these failures equally.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

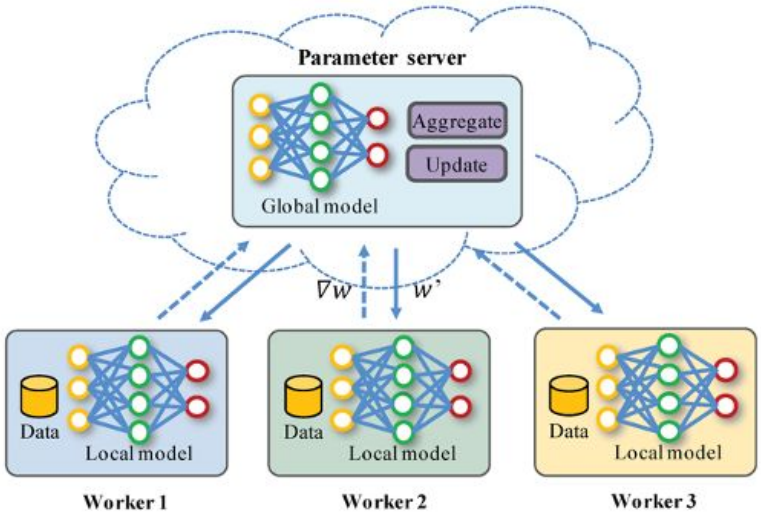
Preliminary work. Under review by the Machine Learning and Systems (MLSys) Conference. Do not distribute.

Background - Deep Learning



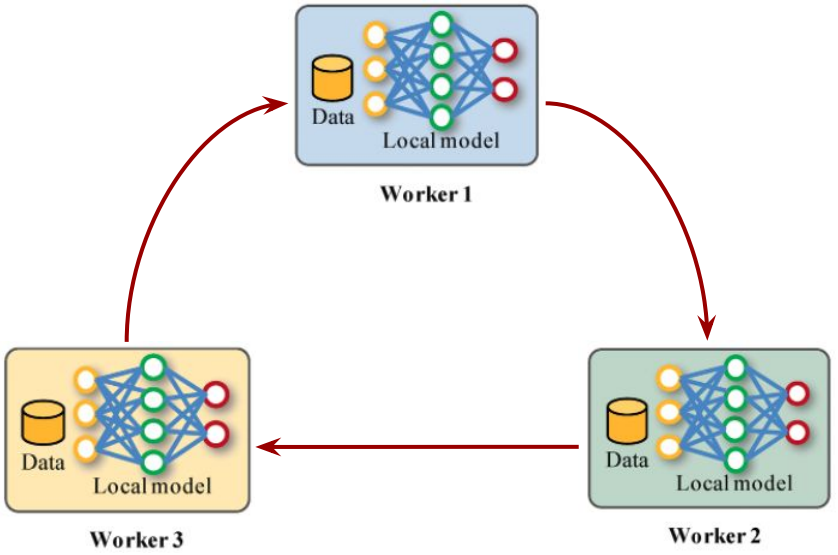
Background - Distributed Deep Learning

Data Parallelism



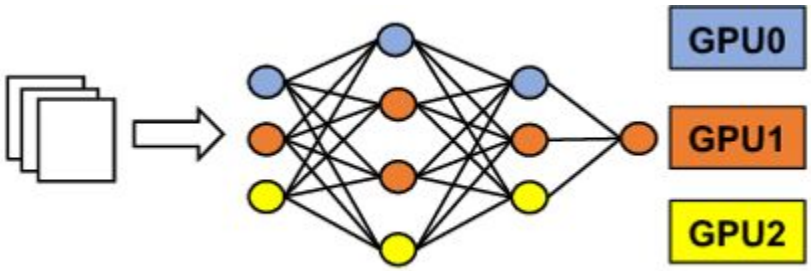
Parameter Server Strategy

Ring (All-reduce)

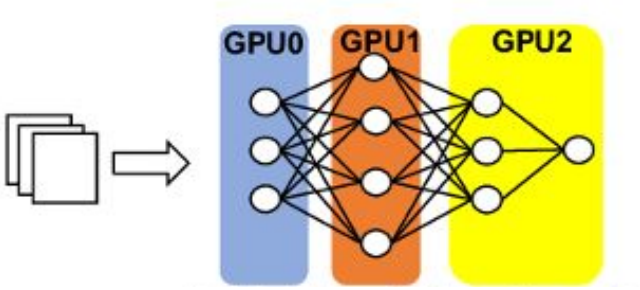


Background - Distributed Deep Learning

Model Parallelism

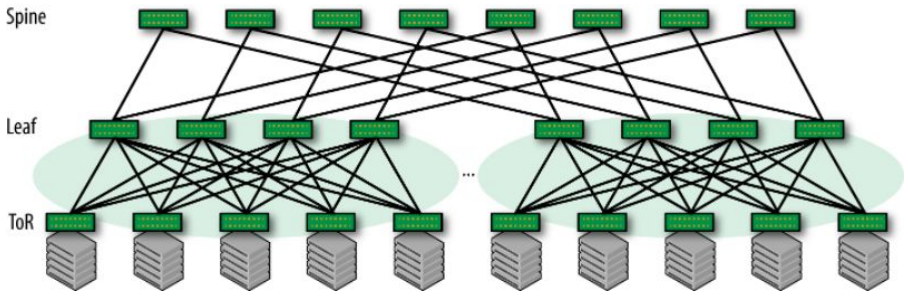


Tensor Parallelism



Pipeline Parallelism

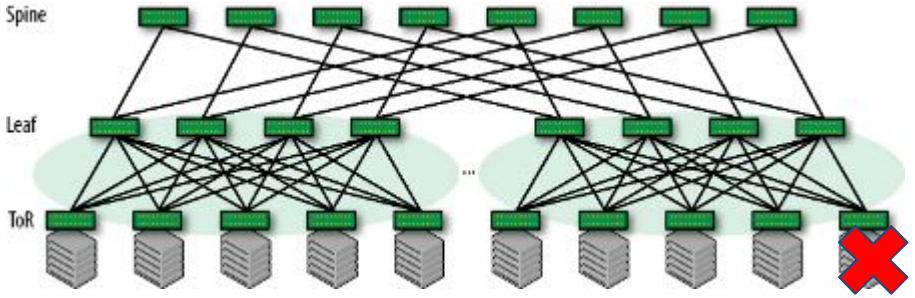
Background - Impact of Failure



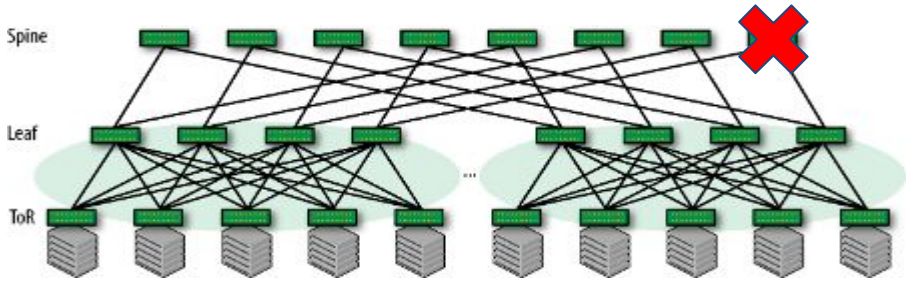
Datacenter Network Architecture

Popular Deep Learning Frameworks:

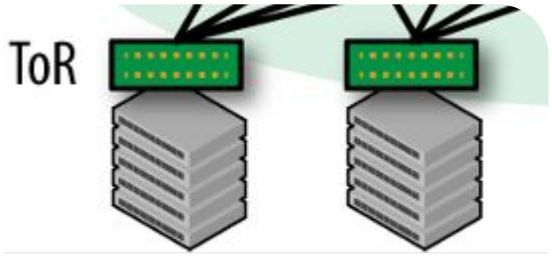
- Take checkpoints every X epochs (Hyperparameter)
- Restart the training from the very recent epoch



Background - Impact of Failure



Failure at network core is handled with minimal disruption!
- Redundancy
- Advance routing protocols



How about network edge?
- ToR switch
- NIC
- Cable in between

=

Host Failure

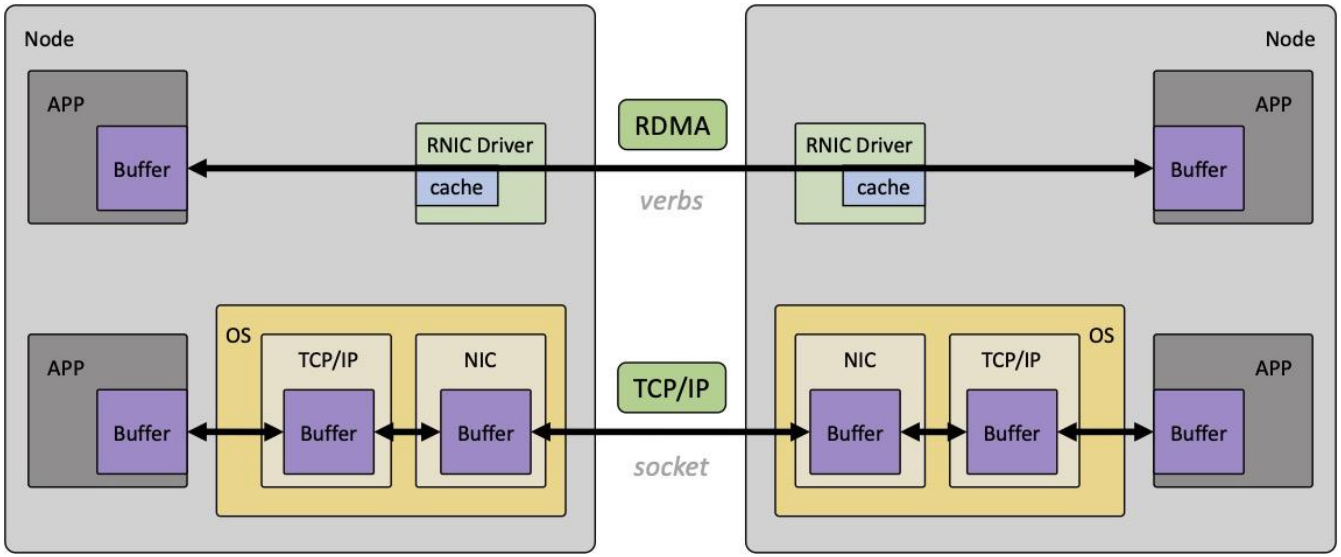
Background - Impact of Failure

Arch	Checkpointing (s)	Model Init (s)	Epoch time (s)
Data parallel	0.77	22.61	4.47
Pipeline parallel	0.53	21.28	8.10
Pipeline & Tensor parallel	0.34	20.91	9.90

Checkpointing and initialization latency/overhead in GPT-2 training on CodeParrot dataset using Megatron-LM

- Average of 1-2 minute disruption (entire cluster) to deal with server failure (at scale of 5 GPUs – higher for larger scale)
- When failures happen, disruptions are costly (e.g., For Meta, a cluster is 32,000 GPUs)
- Network failures account for 8.4% (switches and cables), and 1.7% (NIC) of overall disruptions [10]
- Authors in MegaScale [11]: the init time for Megatron-LM for 2048 GPUs is 1047s => Very Disruptive

Background - RDMA



https://wiki.huangxt.cn/papers/network/RDMA-Related

How to use RDMA

- Get RDMA Devices
- Open a Device
- Register Memory Region
- Setup queue pairs
- Talk!

In this project …

We propose a solution that is

Transparent	⟷	Unmodified application and protocol/hardware
Resilient	⟷	Provide fault detection/recovery mechanism at network edge
Hardware Offloaded	⟷	Maintains RDMA performance and doesn't complicate the network stack

Design

High-level design decisions in THORN to achieve aforementioned goals

Single virtual device that the application can bind to and use RDMA API w/o modification

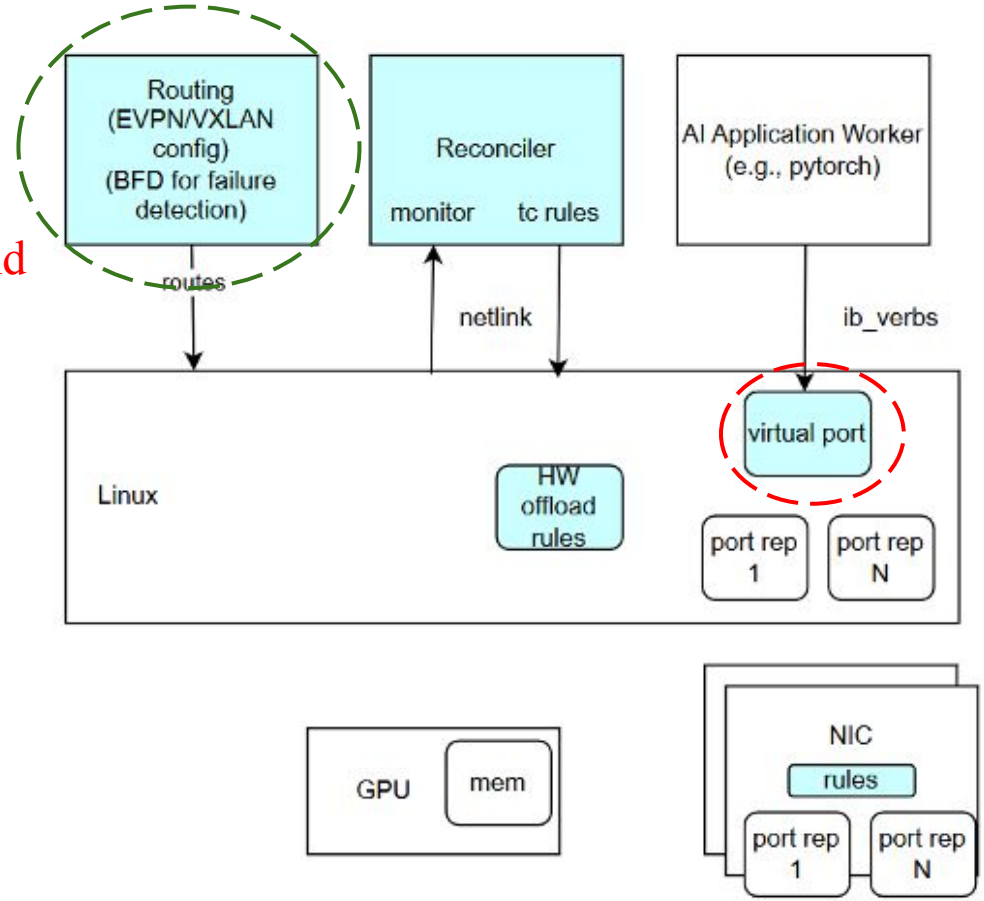
Tunneling the traffic across hosts to hide the underlying network redundancy

Use of standard routing protocols to detect failures and select alternate paths

Architecture Overview

Unmodified Application:

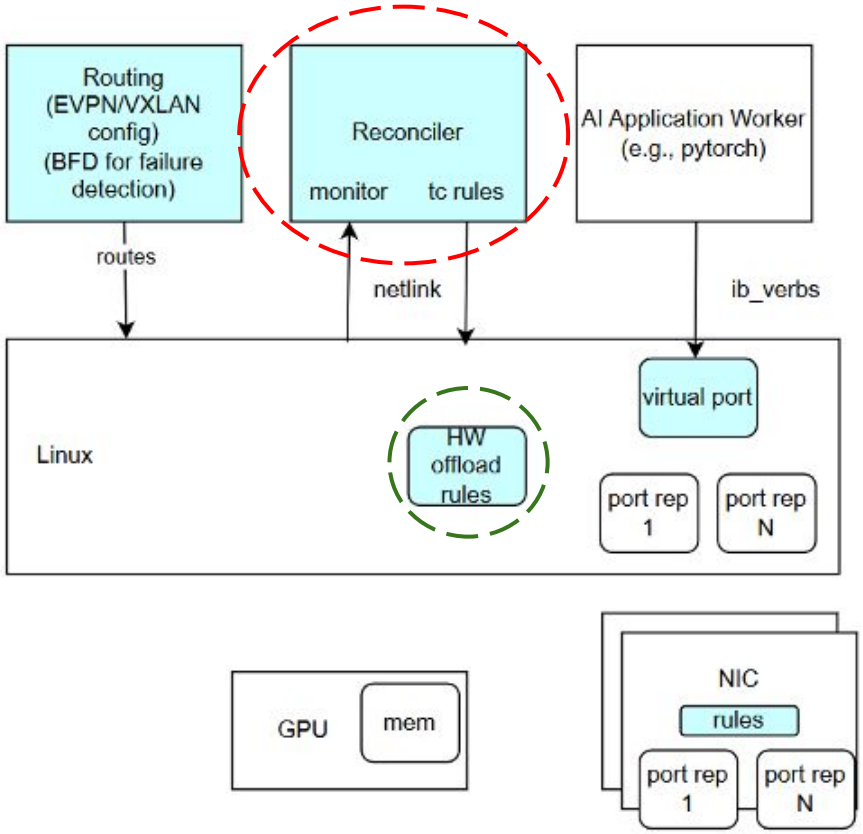
- Single virtual device (SR-IOV) exposed to the application to bind
- Routing software for failure detection and re-routing



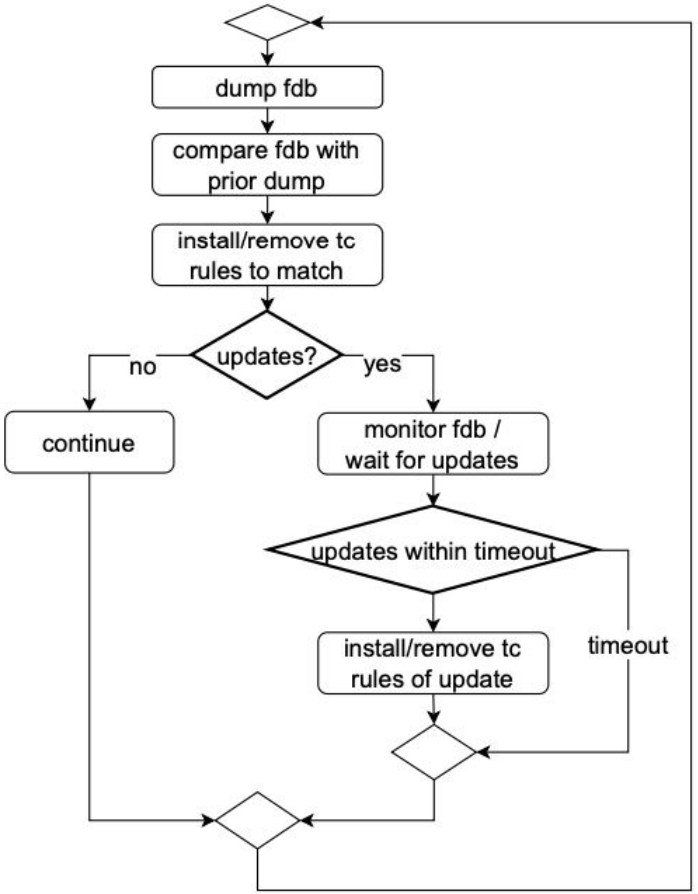
Architecture Overview

Unmodified Hardware

- Reconciler that converts network routes to local rules that direct traffic between virtual port and port representors
- Hardware offload to NIC (programmable)



Architecture Overview: Reconciler



Reconciliation loop for offloading the routing table updates

- Uses *Netlink* to talk to Linux
- Ensures correctness

Evaluation

Testbed Summary:

Cluster Composition: 5 hosts and 2 Mellanox SN2700 switches (48-port, 100Gbps).

Host Configuration:

- Intel Xeon Silver 64-core CPU (2.10GHz).
- Nvidia A100 GPU (16 GB memory).
- Dual-port, 100Gbps Mellanox ConnectX-5 network card (RDMA support).

Network Topology:

- ConnectX-5 card ports connected to separate switches.
- Switches interlinked via an active-backup link pair.

Storage System:

- 1 TB storage node per switch.
- High-throughput, low-latency log collection and dataset distribution.

Macrobenchmark:

ResNet101:

- Dataset: CIFAR10 (60000 32x32 Images)
- Output: 10 Classes
- Framework/Architecture: Pytorch DDP

GPT-2 (Language Model):

- Dataset: CodeParrot (Python Code Generation)
- Framework/Architecture: Megatron-LM
  - Pipeline Parallel
  - Pipeline & Tensor Parallel
  - Data Parallel

NCCL (Nvidia Collective Communication Lib):

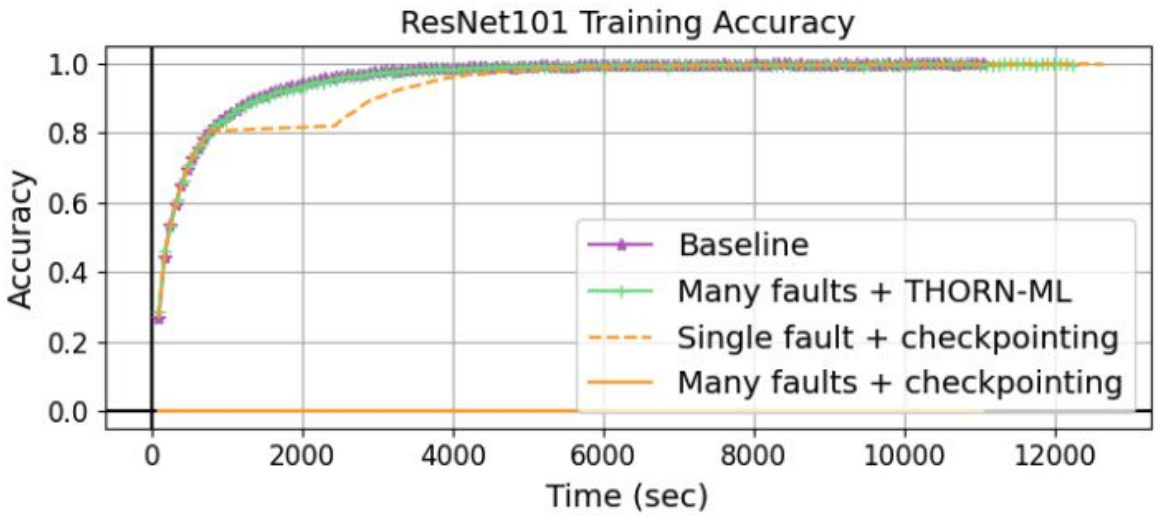
- Operation Bandwidth Measurement

Microbenchmark:

- RDMA performance test (*ib_send_lat*)

Evaluation: Macro-benchmark

THORN is able to keep the overall training job even in extreme fault scenarios w/o noticeable difference with baseline condition



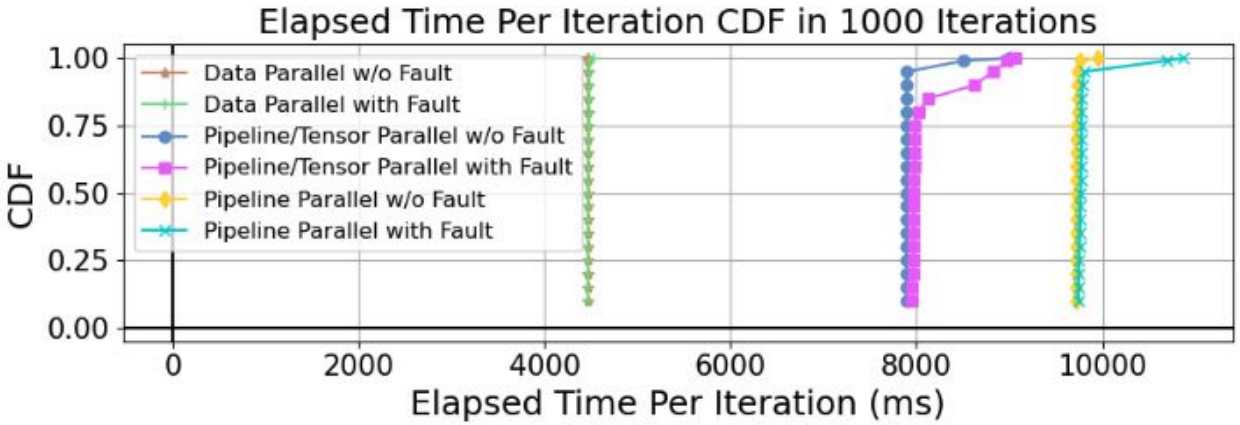
ResNet101 training on CIFAR-10 dataset using Pytorch DDP API on 4 nodes.
Representation of time to accuracy

Note on Single fault with checkpointing:

- Checkpoints happen every 10 epochs
- Accuracy remains the same amount for roughly 13 mins

Evaluation: Macro-benchmark

THORN handles network faults with the overhead of ms scale => order of magnitude less than checkpointing



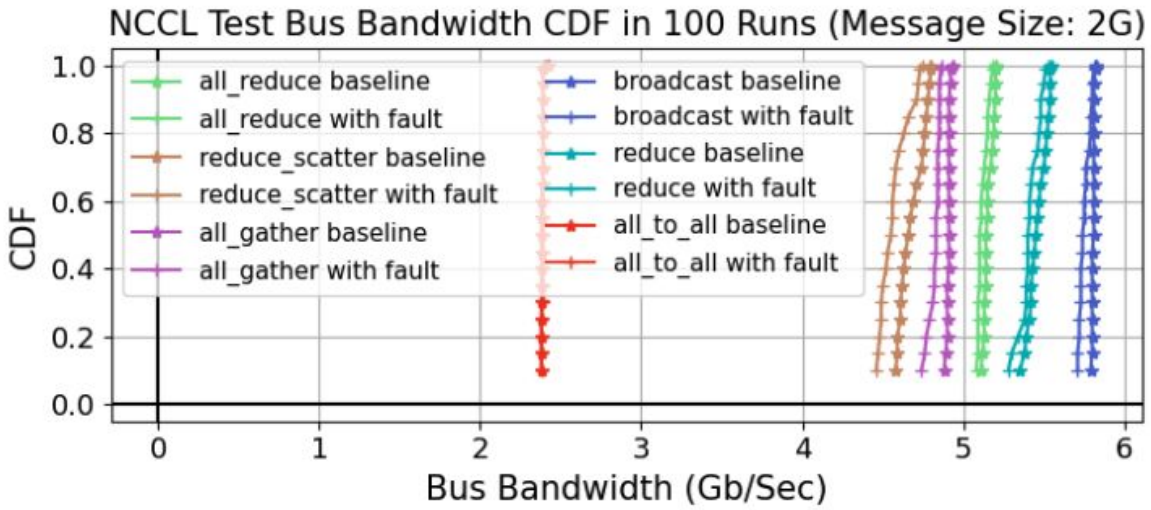
Distributed GPT-2 training on CodeParrot dataset performance. Each data point is averaged over 10 iterations.

Latency Impact Summary:

- **Data Parallel Architecture:**
  - Added latency at the 90th percentile: **5.3ms** (1.1% overhead).
  - Added latency at the 95th percentile: **7.3ms** (1.6% overhead).
- **Pipeline Parallel Architecture:**
  - Added latency at the 90th percentile: **54.9ms** (5.6% overhead).
  - Added latency at the 95th percentile: **76.7ms** (7.8% overhead).
- **Combined Pipeline and Tensor Parallel Architecture:**
  - Added latency at the 90th percentile: **729.4ms** (9.2% overhead).
  - Added latency at the 95th percentile: **928.4ms** (11.7% overhead)

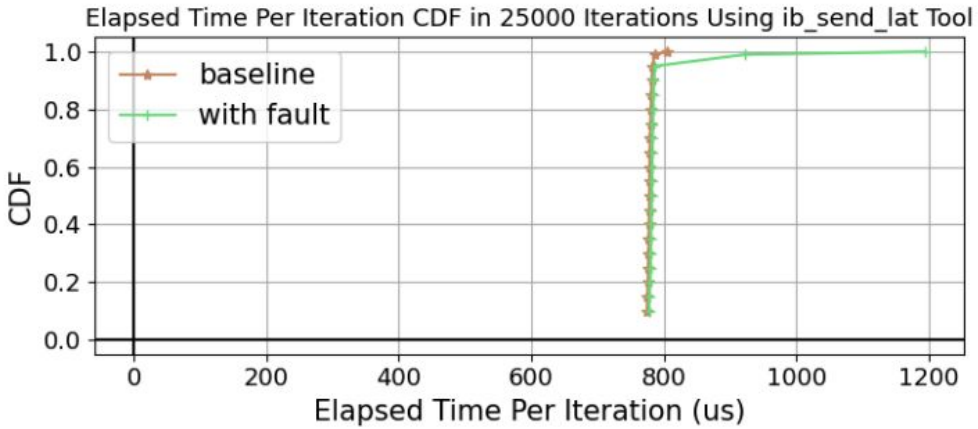
Evaluation: Macro-benchmark

THORN negligible difference (less than 2%) in bus bandwidth of nccl collective operations



- NCCL test collective operations performance report with a message size of 2GB for a baseline without THORN-ML and without faults compared to with THORN-ML and with faults.
  - Similar tests were run with 1GB, 256MB, and 1MB, and each showed that
  - provide nearly the same throughput
- **80th Percentile:** Maximum difference of **0.129Gbps** between fault and non-fault scenarios.

Evaluation: Micro-benchmark



- Using the *ib_send_lat* command with a message size of 2^{23} bytes
- data points with higher delay are due to re-transmission, and the number of packets that need to be re-transmitted indicates the fail-over time
- detection and fail-over time is so small that it only applies in the 99.9th percentile

Related Work

Fast Distributed Deep Learning over RDMA [4]
- Claims that grpc is suboptimal
- Not fault tolerant

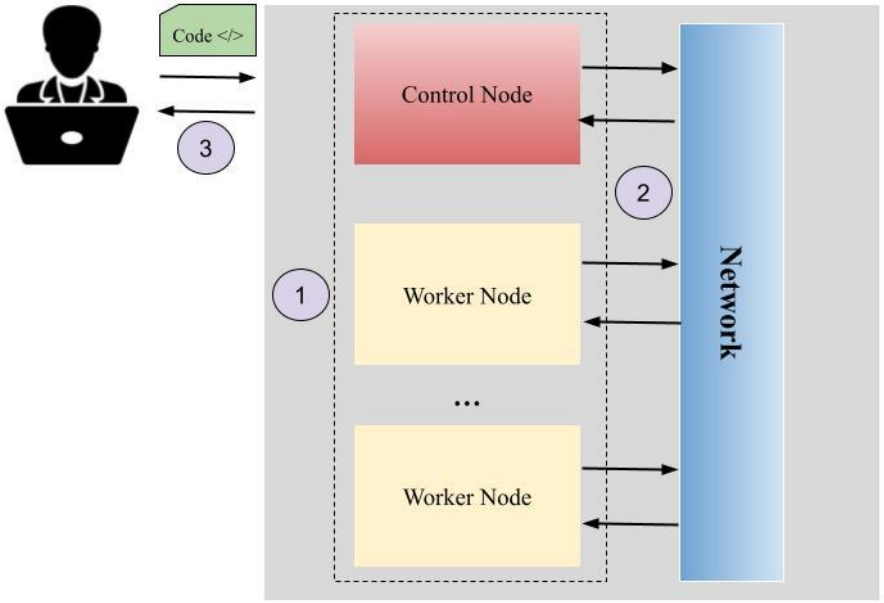
Ultima [5] mitigates the straggler/fault impact in DDL in software level
- Provide fault tolerance at some level
- Not leveraging RDMA connections and hardware offload

Popular frameworks like Tensorflow handle faults in software level using checkpointing mechanism
- High overhead

Maestro [6] Proposes multipath RDMA for DDL workload at datacenters
- Implemented in user-space/software
  - Modified Application
  - Low performance

MP-RDMA[12] handles congestion in RDMA networks (address ECMP problem)
- Uses specialized hardware (modified hw/protocol)
- not applicable on the edge of the network

Recap



1. Escra, Event-driven Sub-second Container Resource Allocation
2. THORN-ML: Transparent Hardware Offloaded Resilient Networks for RDMA based Distributed ML Workloads
3. Optimizing Cloud Configuration Selection for Cost-Effective Distributed Deep Learning Execution

Optimizing Cloud Configuration Selection for Cost-Effective Distributed Deep Learning Execution

[In submission to ICML 2025]

Everybody uses Deep Learning
- Including non-expert users

Distributed Deep Learning is Popular
- Model and Data is getting larger

Numerous options are available in cloud platform
- e.g. Tens of HW Configs

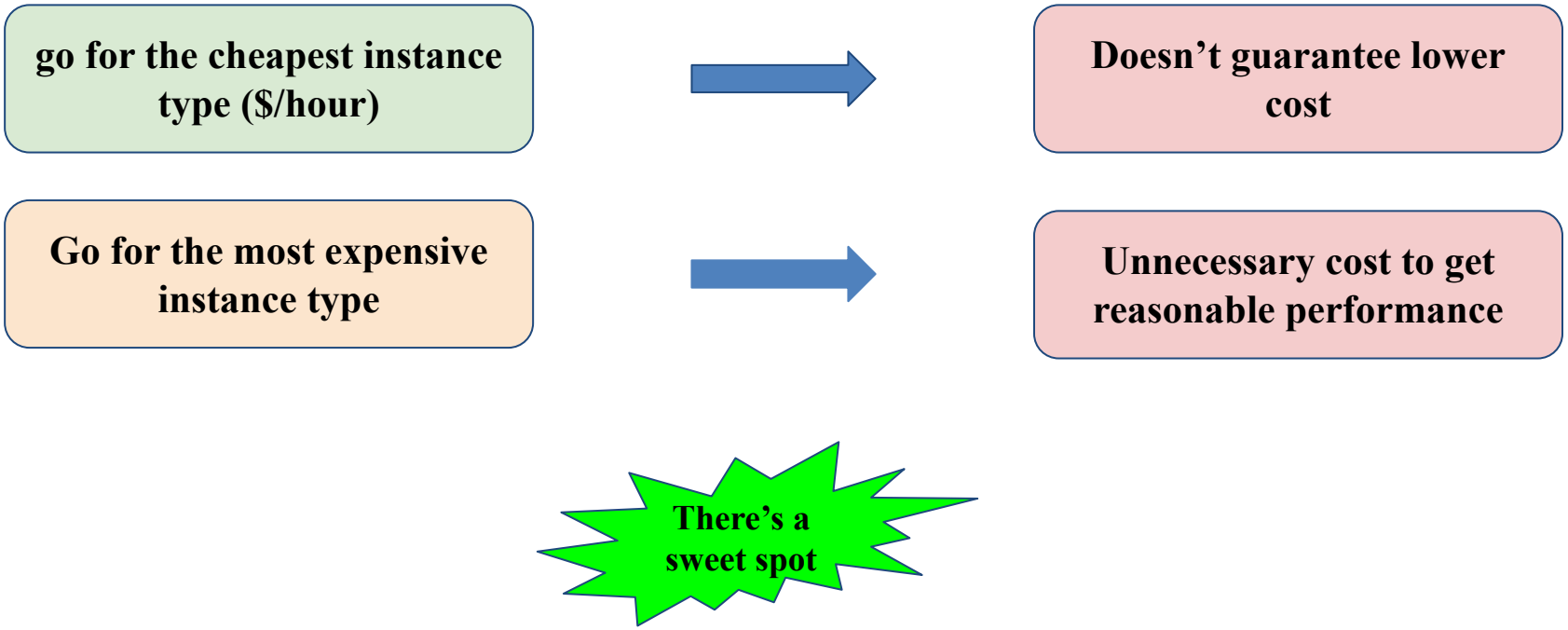
Problem Statement

How to choose the best configuration in the cloud?

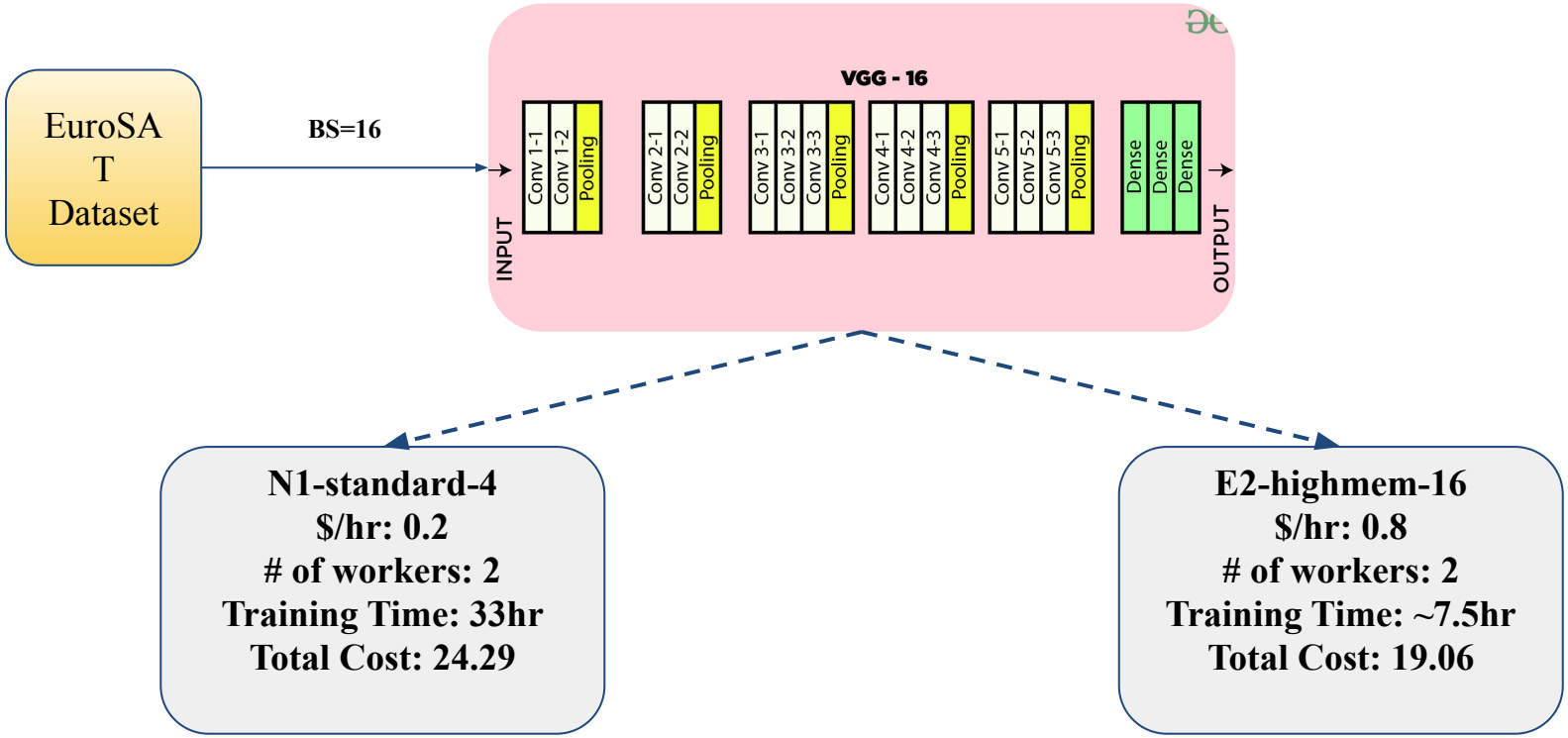
which one is more expensive and which one is faster to run my training task?

Problem Statement

Naive solution leads to suboptimal results!



Motivating Example



Background Work

CherryPick [8]

- Leverage Bayesian Statistics to predict the best cloud configurations
- Still needs a few sample runs in the cloud
  - ***mostly works for recurrent jobs***
- Not tailored for Distributed Deep Learning

Ernest [9]

- Creates resource consumption model of machine learning jobs running on
- Needs retraining for every given VM type
- Not tailored for Distributed/Deep Learning

None of the previous work directly address the problem of cost and training time prediction in distributed deep learning platforms

Our Approach

In many cases, the training task can run partially on a local weak setup (i.e laptop)!

Why don't we leverage a minimal local run as a representative workload and predict the cost and training time on a powerful cloud?

Our Approach

Deep Learning based solution that

- Input
  - Partial resource usage on local setup
  - Training job features such as parameters/hyperparameters
  - Target desired configuration and setup in the cloud (e.g. 3 workers using e2-highmem-16 instance)
- Predict
  - training time and cost for specific configuration

Compare the performance with traditional machine learning algorithms

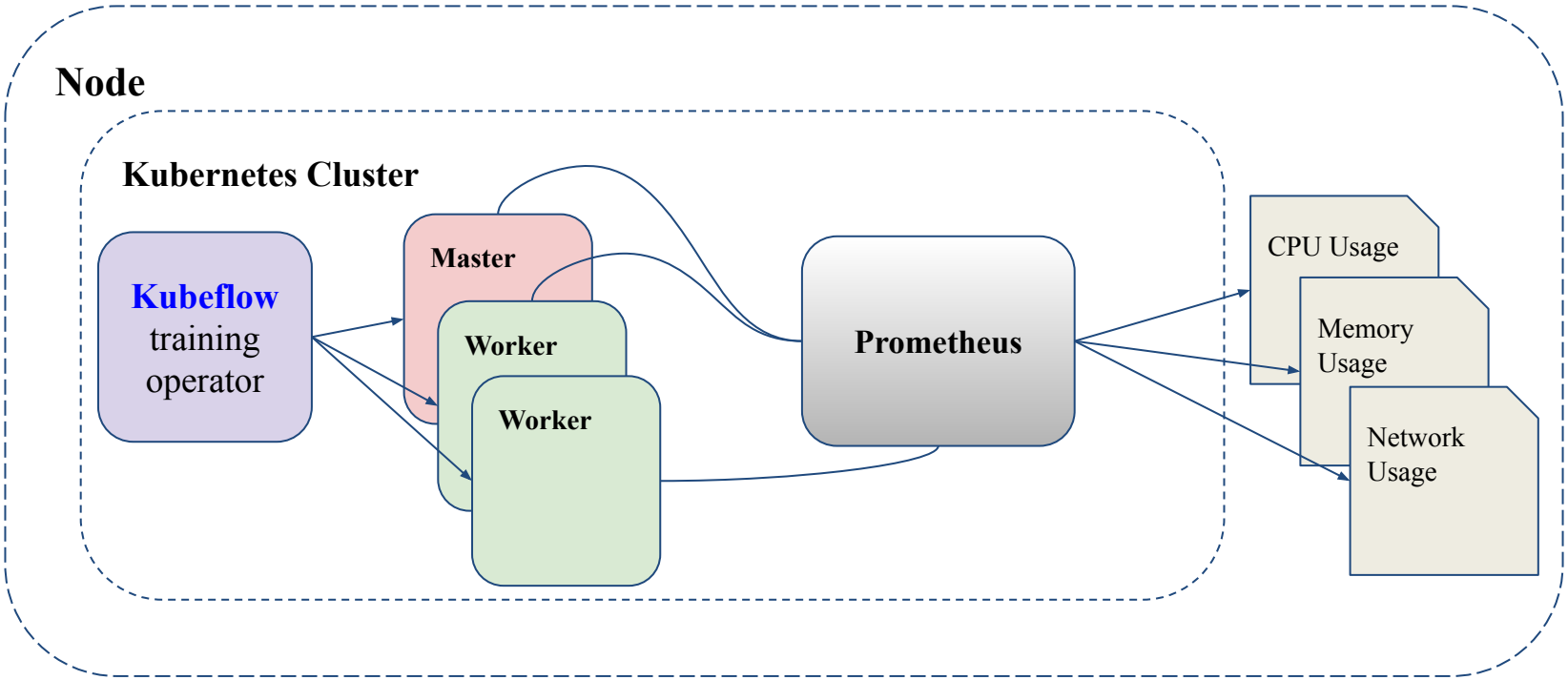
Let's hope the user make informed decisions with this results

Data Collection

Local

- Represents a partial run on a local laptop (single machine)
- All jobs are run in a 3-worker architecture (1 master, 2 workers)
- Collected partial usage data on 9 different deep learning tasks
  - Resource usage time series
    - ***cpu usage***
    - ***memory usage***
    - ***inbound network usage***
    - ***and outbound network usage***

Partial Training Job Data Collection



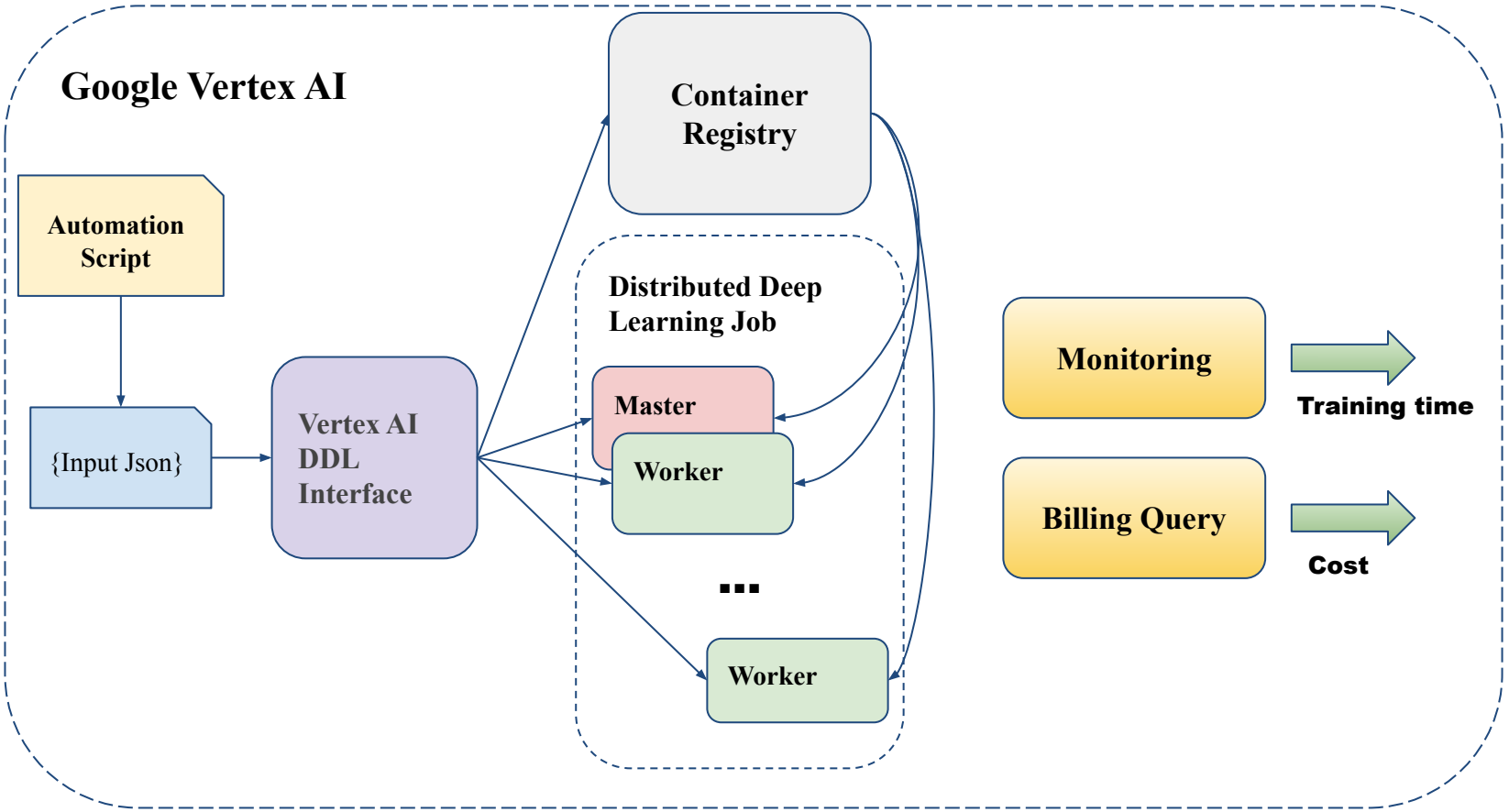
Data Collection

Google Vertex AI

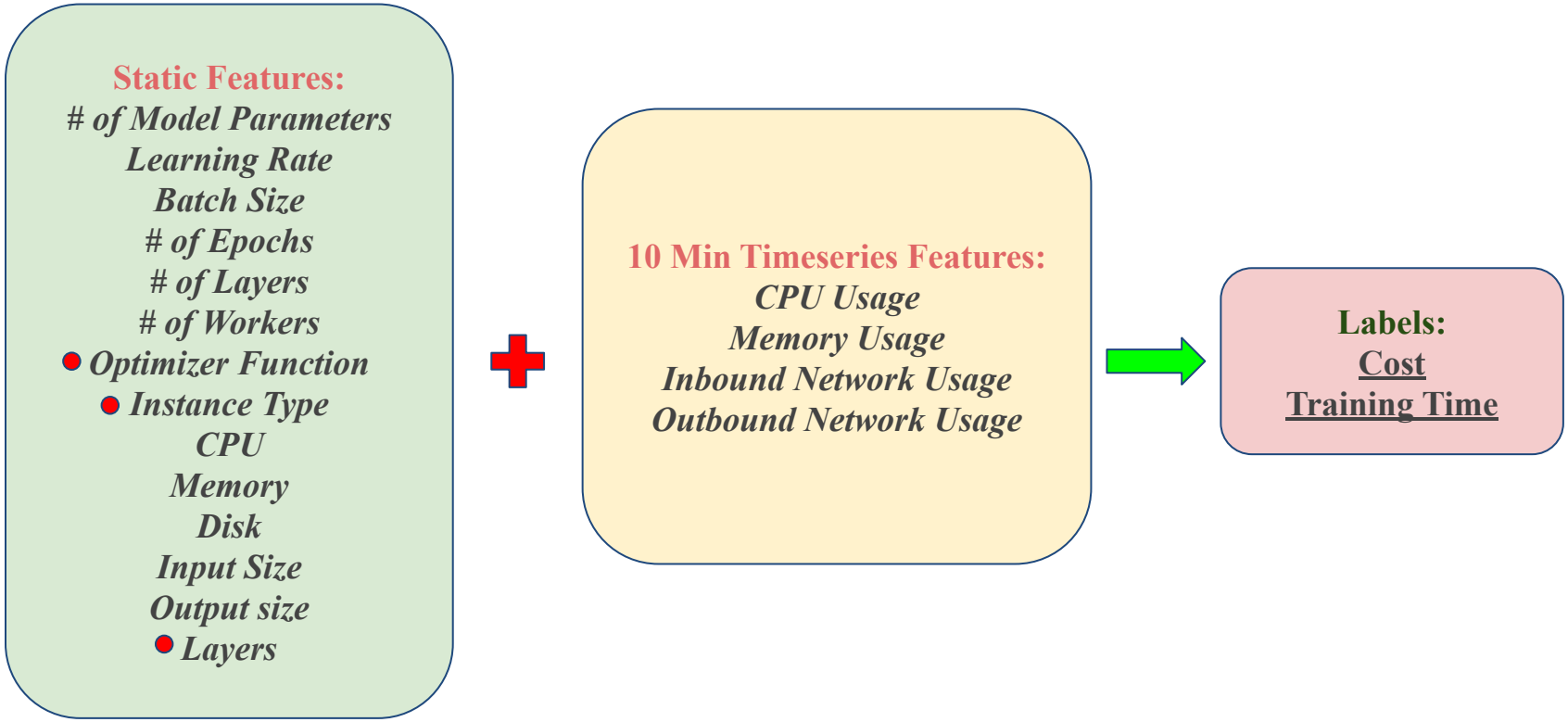
- GCP service offering to run distributed deep learning tasks easily using custom docker containers
- Run each deep learning job with several different configurations
  - Change Model hyperparameters (e.g. *batch size*)
  - Change Resource configuration (e.g. *instance type* & *number of workers*)
- Record cost and training time for every training job

Training Job Data Collection

Dataset Collection:

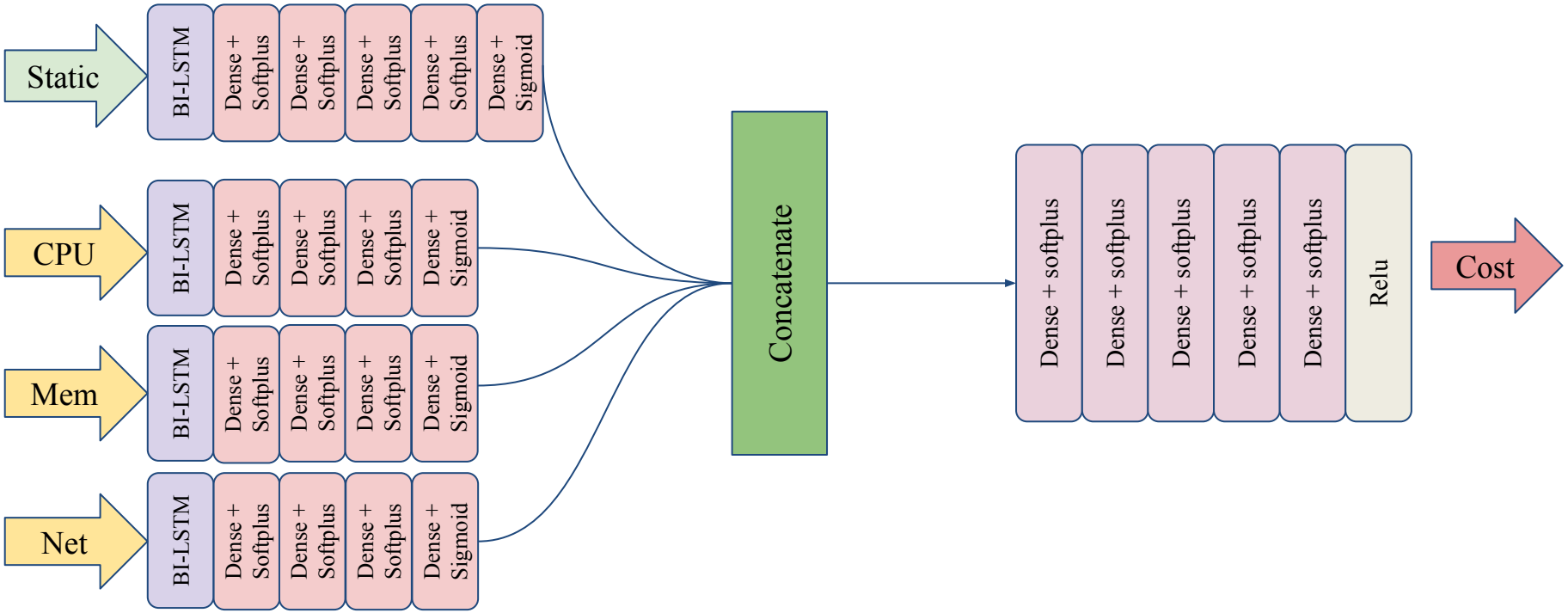


Features and Data Preprocessing



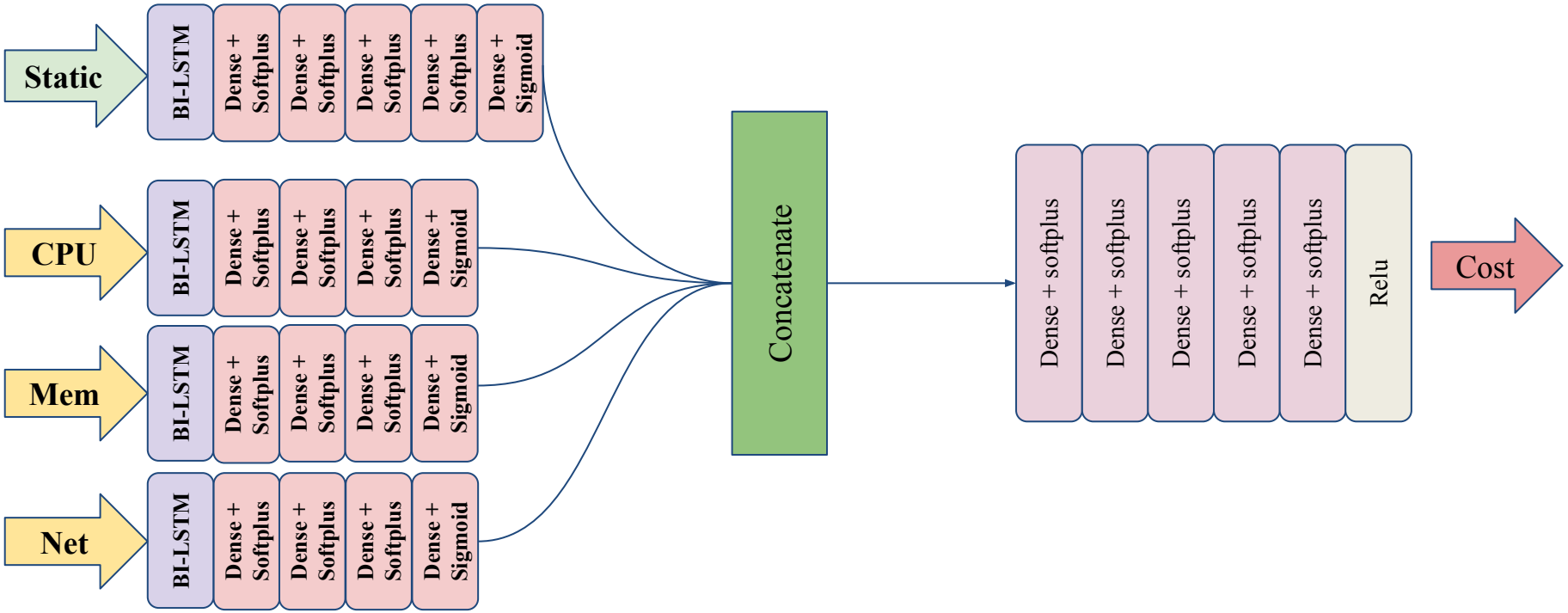
Deep Learning Model Design

Cost Model Development:



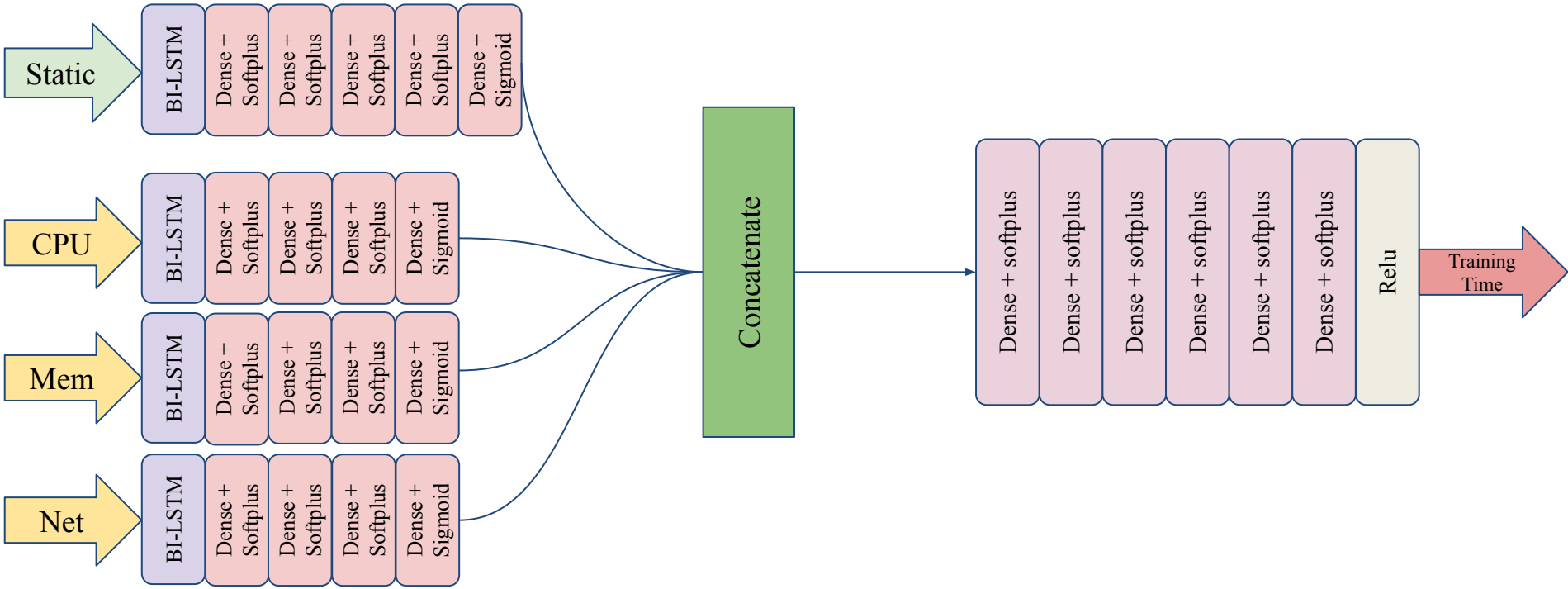
Deep Learning Model Design

Cost Model Development:



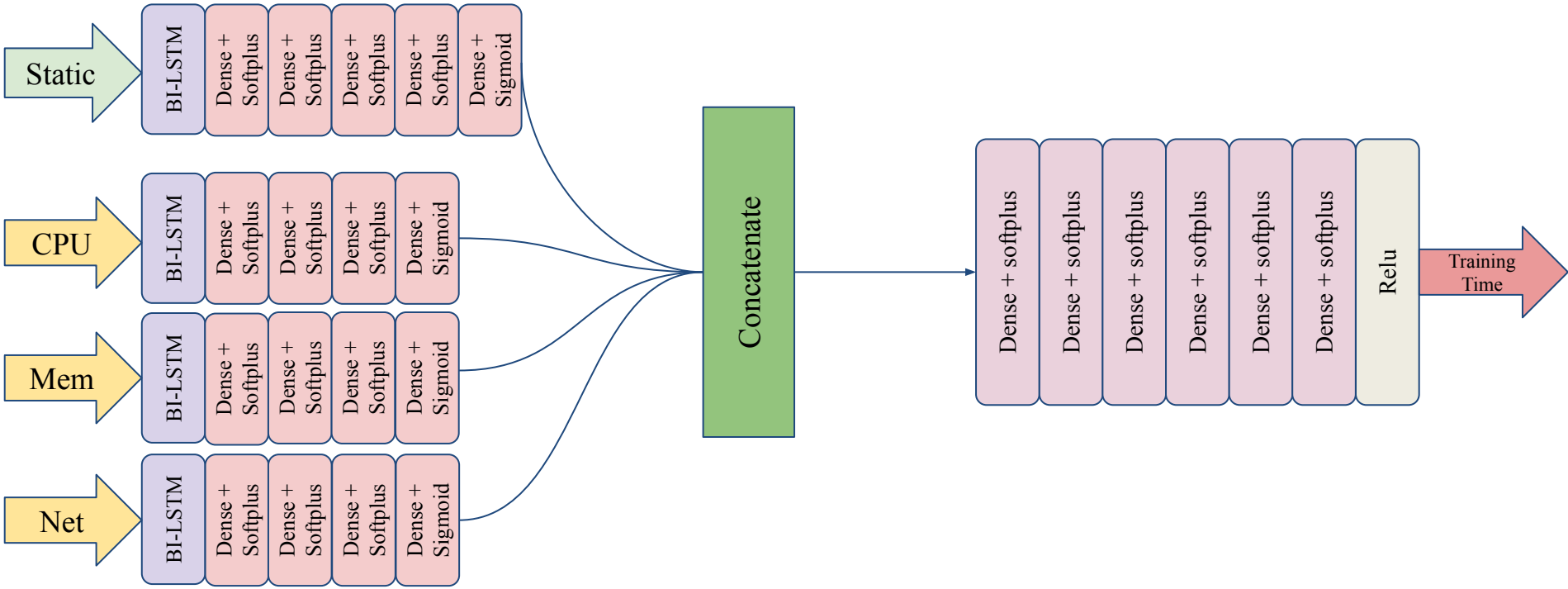
Deep Learning Model Design

Training Time Model Development:



Deep Learning Model Design

Training Time Model Development:

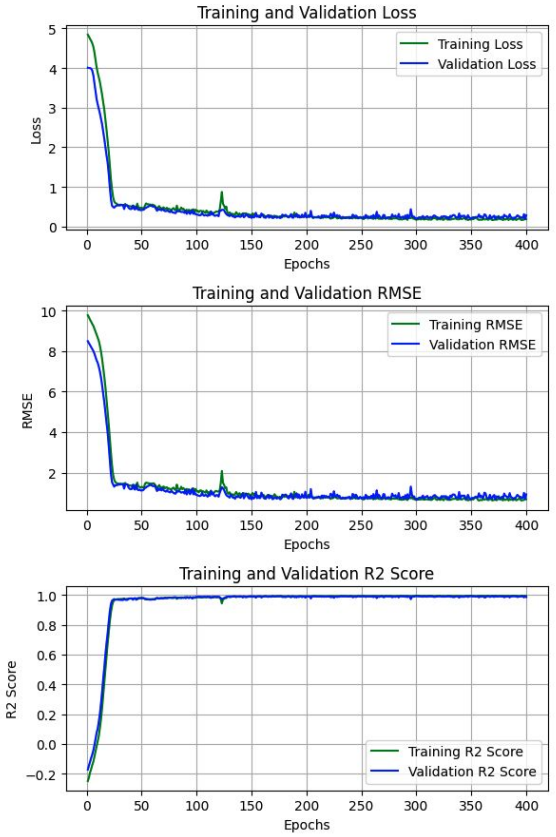


Deep Learning Training Performance (Cost)

- No signs of abnormality in training
- Converge to acceptable RMSE (11% / 13% of the average)
- Achieving R2-score close to ~1 shows a pretty good fit!

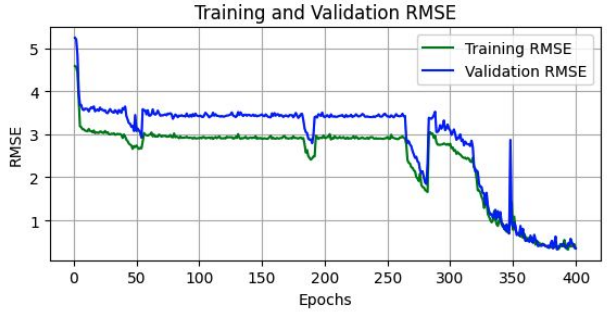
Model Hyperparameters:
Loss: Huber (delta=1.0)
Optimizer: Adam (lr=0.0001, beta=0.5)
Epochs: 400
Batch Size: 32
Train/Validation Split: 80%/20%

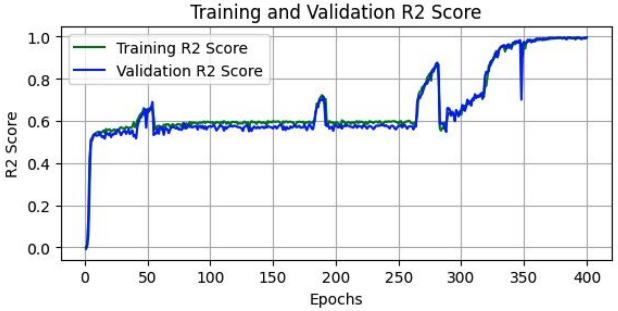
	Training	Validation
Loss	0.1175	0.2046
RMSE	0.5602	0.7668
R2 Score	0.9959	0.9904



Deep Learning Training Performance (Training Time)

- No signs of abnormality in training
- Converge to acceptable RMSE (16% / 13% of the average)
- Achieving R2-score close to ~1 shows a pretty good fit!



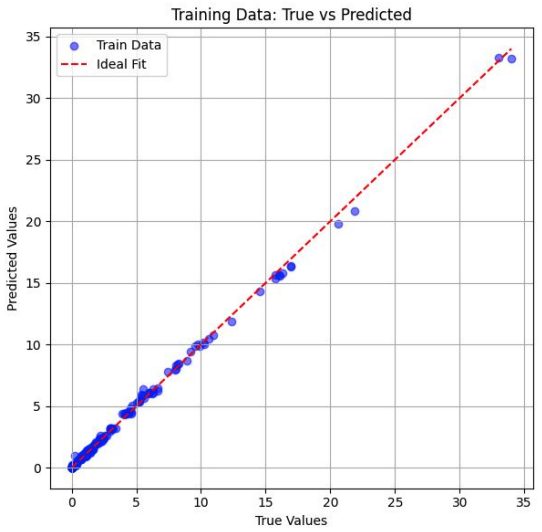


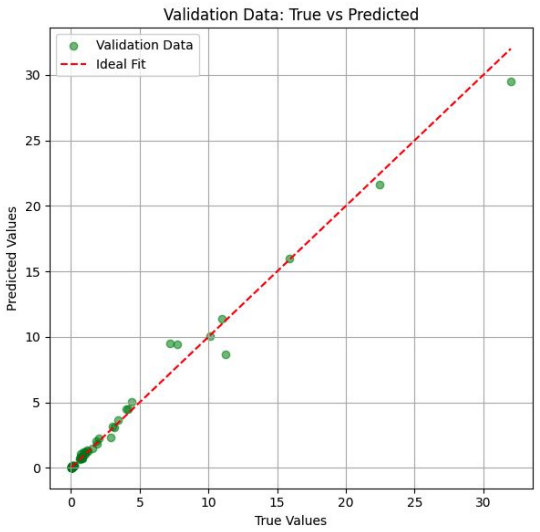
Model Hyperparameters:
Loss: Poisson
Optimizer: Adam (lr=0.0008, beta=0.5)
Epochs: 400
Batch Size: 32
Train/Validation Split: 80%/20%

	Training	Validation
Loss	-2.2585	-2.6542
RMSE	0.4095	0.3273
R2 Score	0.9920	0.9961

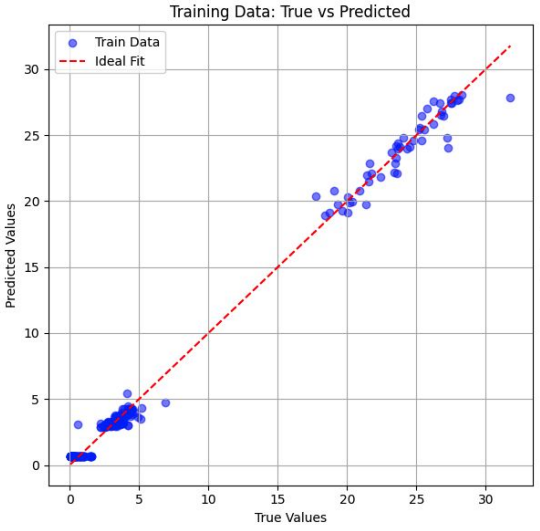
Confusion Plots vs Perfect Fit

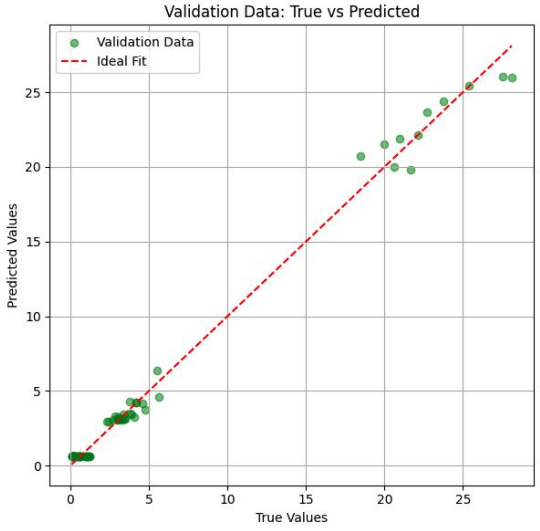
Training Time confusion plot





Cost confusion plot

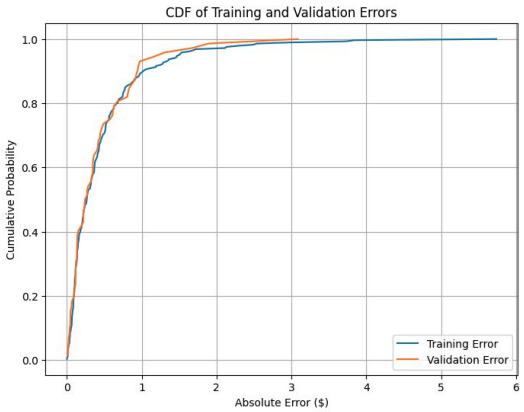


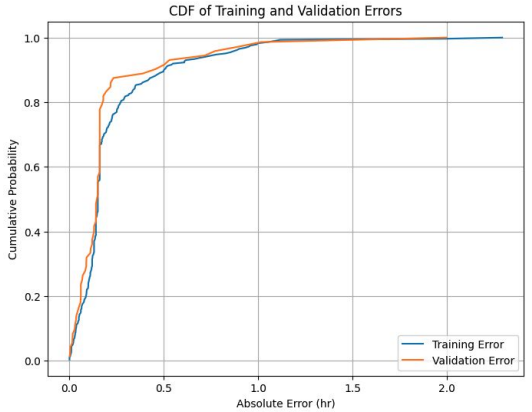


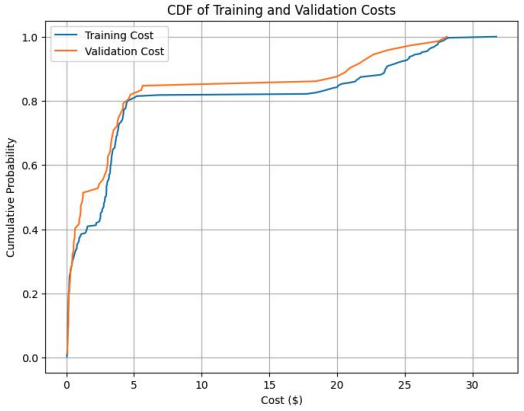
Error Distribution

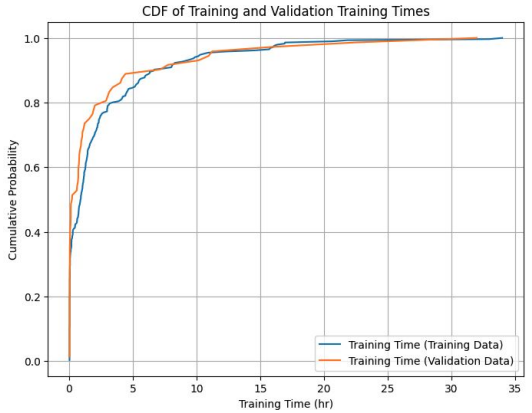
- Cost Value 90 %ile: 23.6$ and 20.9$,
- Prediction Error 90 %ile: 0.78$ and 0.93$

- Training time 90 %ile: 6.65 and 6.89 hr
- Prediction Error 90 %ile: 1.5 hr









Deep Learning vs Traditional ML Algorithms

ML Algorithms:

- Linear Regression/Poisson Regression for Cost/Training Time prediction ✖
  - Not showing promising results compared to DL
- Random Forest Regression
- XGBoost

Feature Engineering:

- Random Forest Top 5 features
  - **Cost Model:**
    - Partial resource usage
    - An increase in resource usage during the later stages of the time series correlates with a higher cost.
  - **Training Time Model:**
    - Increase in the # of workers/machines correlates with decrease in training time
    - More CPU correlates with less training times
    - n1-standard-4 instance has a wider range in predictions

Machine Learning Models Comparison (refer to thesis)

Random Forest Regressor and XGBoost show comparable performance to deep learning

	Cost Prediction		Training Time Prediction	
Linear/Poisson Regression	Training	Validation	Training	Validation
RMSE	1.43	1.34	0.28	0.31
R2 Score	0.97	0.97	0.96	0.96
Random Forest Regressor	Training	Validation	Training	Validation
RMSE	0.33	0.70	0.28	0.34
R2 Score	0.99	0.99	0.996	0.995
XGBoost	Training	Validation	Training	Validation
RMSE	0.19	0.70	0.08	0.26
R2 Score	0.999	0.99	0.999	0.997

Deep Learning vs Traditional ML Algorithms

Deep Learning

- Dataset Size and Complexity
- Automated Feature Engineering
- Transfer Learning
- Sequential Data
- Computationally intensive
- Generalize well!

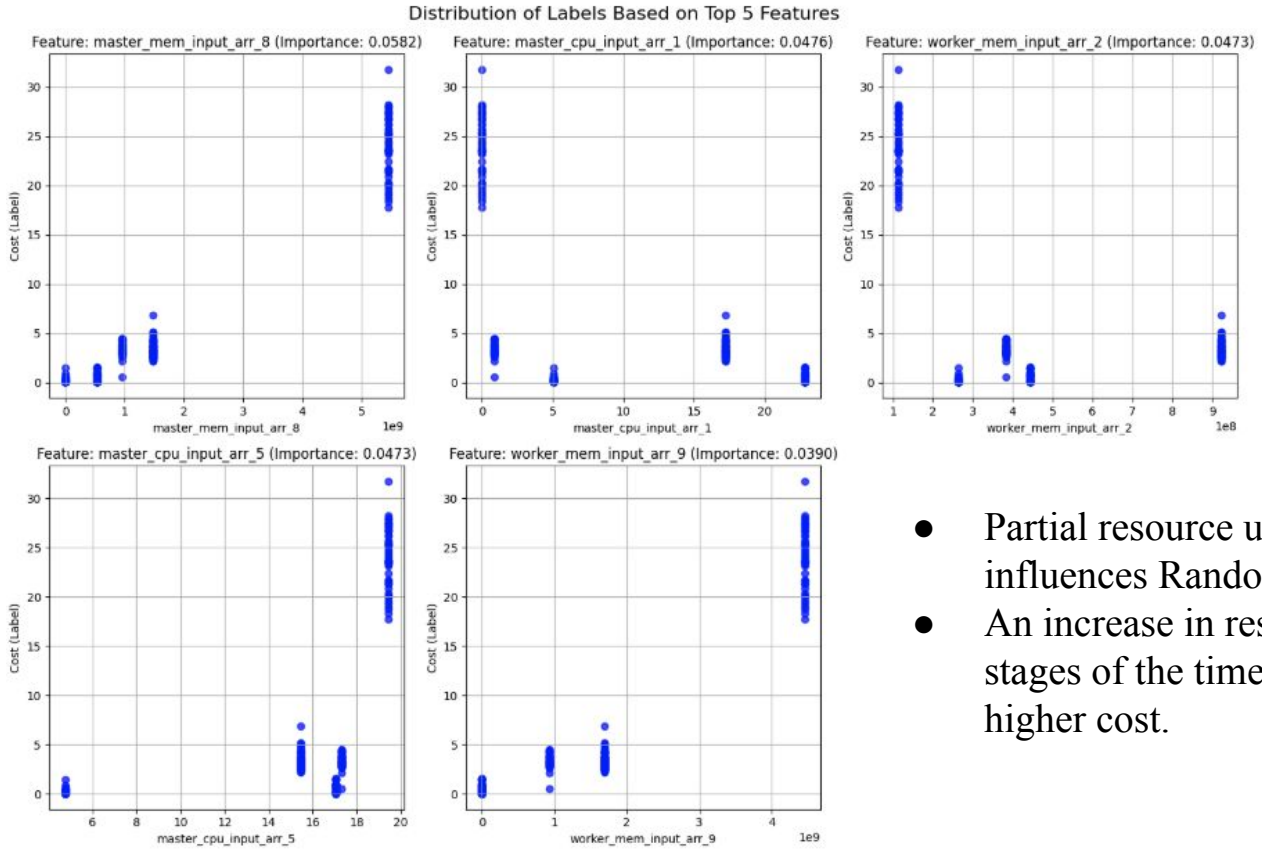
ML Algorithms:

- Less computationally intensive
- Cannot naturally handle sequential data
- Need manual feature engineering
- Lose to DL when it comes to large scale tasks and complex datasets
- ALgorithms like XGBoost usually appear to overfit the training data

Hybrid Methods:

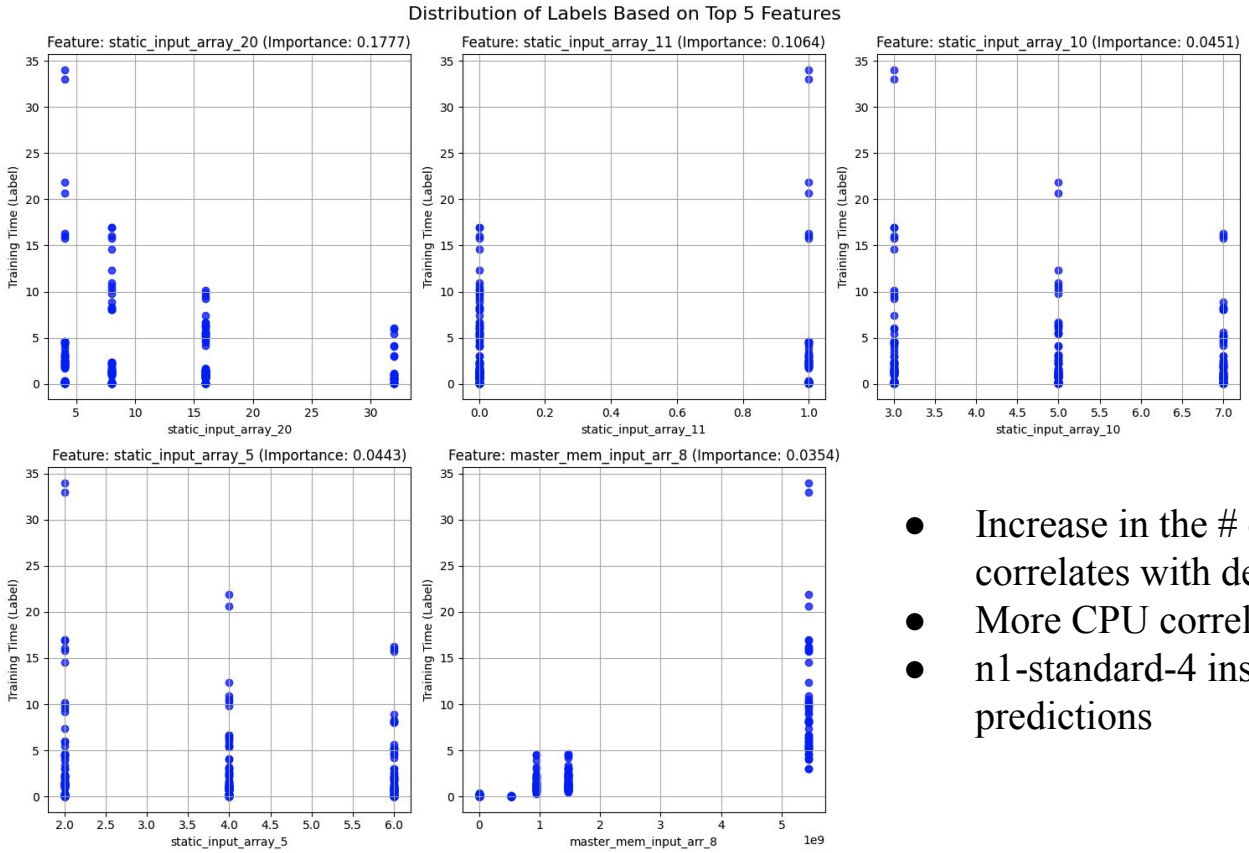
- Model Stacking
- Ensemble Methods

RF model feature importance analysis



- Partial resource usage significantly influences Random Forest predictions.
- An increase in resource usage during the later stages of the time series correlates with a higher cost.

RF model feature importance analysis



- Increase in the # of workers/machines correlates with decrease in training time
- More CPU correlates with less training times
- n1-standard-4 instance has a wider range in predictions

Static_input_arr_11: n1-standard-4 instance type
Static_input_arr_10: # of Machines
Static_input_arr_20: Available CPU
Static_input_arr_5: # of workers

Conclusion

In this thesis:

- We studied the advantages of fine-grained resource allocation in containerized environments
- And proposed a fast resilient network architecture for state-of-the-art distributed deep learning workload
- Finally, we investigated distributed deep learning cost and training time predictions in the cloud to help DDL developers make informed decisions on choosing their cloud configurations

- Minimizing waste:
  - In overprovisioning through efficient auto-scaling
  - of entire clusters for extended periods due to isolated network edge failures
  - Resources in instance selection, preventing overuse of large instances when smaller, cost-effective options can suffice

Thank You!

References

[1] Google Vertex AI
[2] VPA
[3] Autopilot
[4] Fast DDL over RDMA
[5] Ultima
[6] Maestro
[7] tc-flower
[8] CherryPick
[9] Ernest
[10] Llama3
[11] MegaScale
[12] MPRDMA