Efficient and Resilient Distributed Deep Learning for Cloud-based Execution

by

Maziyar Nazari

B.Sc., University of Tehran, 2018

M.Sc., University of Colorado Boulder, 2020

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2025

Committee Members:

Eric Keller, Chair

Eric Rozner

Shivakant Mishra

Tamara Silbergleit Lehman

Qin Lv

Nazari, Maziyar (Ph.D., Computer Science)

Efficient and Resilient Distributed Deep Learning for Cloud-based Execution

Thesis directed by Prof. Eric Keller

Machine and deep learning applications have become increasingly popular among developers, engineers, and researchers. However, the rapid evolution of deep learning models and datasets presents significant challenges for both users and cloud providers in initiating and executing machine/deep learning tasks on cloud infrastructure. Key issues, including the complexities of selecting optimal cloud configurations, resource efficiency, latency, throughput, and overall performance, have become critical factors in the successful deployment of machine learning applications on cloud platforms.

This thesis explores the existing gaps in the state-of-the-art execution pipelines for machine and deep learning tasks on the cloud, which often result in suboptimal performance across the aforementioned metrics. The objective is to address the challenges associated with selecting the right cloud configurations, improving resource efficiency, and achieving superior outcomes in terms of latency, throughput, and overall performance when deploying machine learning applications on cloud services.

Dedication

To my family: Sepideh, Mom, and Dad. I love you. Thank you for all of the time you put in to make me succeed. I owe everything to you and it wouldn't have been possible without your support and kindness.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my beloved wife, Sepideh. Your unwavering love, patience, and support have been my constant source of strength throughout this journey. Your encouragement, understanding, and sacrifices have made this achievement possible, and I cannot thank you enough for always being there for me, both in moments of doubt and triumph.

I am also immensely grateful to my Mom and Dad, for their boundless love, guidance, and support. You have always believed in me, and your encouragement has been a driving force in my academic and personal growth. Thank you for your sacrifices and for providing me with the values of perseverance and dedication that have shaped who I am today.

To my brothers, Farshid & Farzad, thank you for your constant support, camaraderie, and encouragement. Your understanding and the fun moments we shared have been invaluable, and I am grateful to have you by my side.

Finally, I would like to express my sincere thanks to my advisor, Dr. Eric Keller, for your mentorship and guidance throughout my research. Your expertise, insightful feedback, and constant encouragement have played a crucial role in shaping this thesis. I have learned so much from your guidance, and I am truly grateful for the opportunity to work under your supervision.

Contents

Tables

Table

Figures

Figure

Chapter 1

Introduction

Deep Learning applications have surged in popularity, transcending the boundaries of computer science to captivate professionals in diverse scientific and engineering fields. This transformative technology has revolutionized machine learning by enabling the integration of larger models and harnessing extensive datasets, as underscored by recent research findings. Yet, the deployment of machine/deep learning applications presents multifaceted challenges, impacting both the user/developer and infrastructure sides.

Developers face numerous challenges throughout the AI development lifecycle. These include the steep learning curve of machine learning (ML) model development, the complexities of hyperparameter tuning, and the necessity of understanding distributed systems for deep learning at scale. On top of these, selecting the appropriate cloud configurations and hardware for training ML models—especially custom distributed deep learning models—can be particularly daunting.

At the same time, a significant portion of datacenter workloads now comprises machine learning and deep learning tasks, including model training. This shift presents new challenges for cloud providers, who must efficiently allocate and schedule these tasks across available resources while optimizing critical metrics such as latency, resource utilization, quality of service, and fault tolerance.

Addressing these challenges requires a comprehensive understanding of machine and deep learning application development, particularly concerning critical system metrics such as latency, resource utilization, quality of service, and fault tolerance. This thesis aims to bridge these gaps by proposing novel solutions and providing insights for more efficient deployment of machine and deep learning applications in modern computing environments.

Machine and deep learning tasks are increasingly hosted on cloud infrastructure. Although virtualization technologies such as virtual machines (VMs) and containers offer flexible options for deploying these workloads, recent benchmarks and analyses indicate significant inefficiencies in resource allocation and management. These inefficiencies often lead to resource wastage and increased latency. Focusing specifically on cluster management (illustrated in Figure 1), Chapter 2 extends the state-of-the-art in automated resource allocation within containerized environments.

In this chapter, we introduce Escra, a next-generation container orchestrator designed for fine-grained, event-driven resource allocation. Escra supports both single-container and distributed resource management across multiple worker nodes. By performing resource allocation at sub-second intervals both within and across hosts, Escra enables operators to scale resources dynamically and seamlessly without incurring performance penalties. Our evaluation of Escra in microservices and serverless environments demonstrates significant improvements, including reduced application latency, increased throughput, and minimized resource wastage.

Beyond resource management, distributed deep learning (DDL) frameworks such as TensorFlow and PyTorch increasingly rely on Remote Direct Memory Access (RDMA) networks for efficient inter-node communication. RDMA facilitates high-bandwidth, low-latency communication by bypassing the kernel, making it a critical enabler of collective operations like All-Reduce in distributed algorithms such as Stochastic Gradient Descent (SGD). However, these environments are vulnerable to network faults and node failures, which are common in distributed settings.

In Chapter 3, we investigate the impact of network faults and failovers on distributed deep learning workloads and propose a transparent, hardware-offloaded, fault-tolerant communication mechanism. This mechanism is designed to operate seamlessly across DDL workers, mitigating the effects of network faults without requiring modifications to applications, hardware, or underlying protocols. Our solution maintains the performance advantages of RDMA while enhancing resilience, ensuring reliable operation even under challenging network conditions.

Finally, as datasets and models continue to grow in size, distributed deep learning and cloud-based training have become increasingly prevalent. A major challenge for developers lies in selecting optimal or near-optimal cloud configurations for their workloads. In the final chapter, we address this issue by leveraging insights from lightweight, localized runs of deep learning tasks. By analyzing resource usage and application behavior, we develop predictive models for two critical metrics: cost and training time. These predictions empower users to make informed decisions about cloud configurations, balancing efficiency and cost-effectiveness for large-scale deep learning workloads.

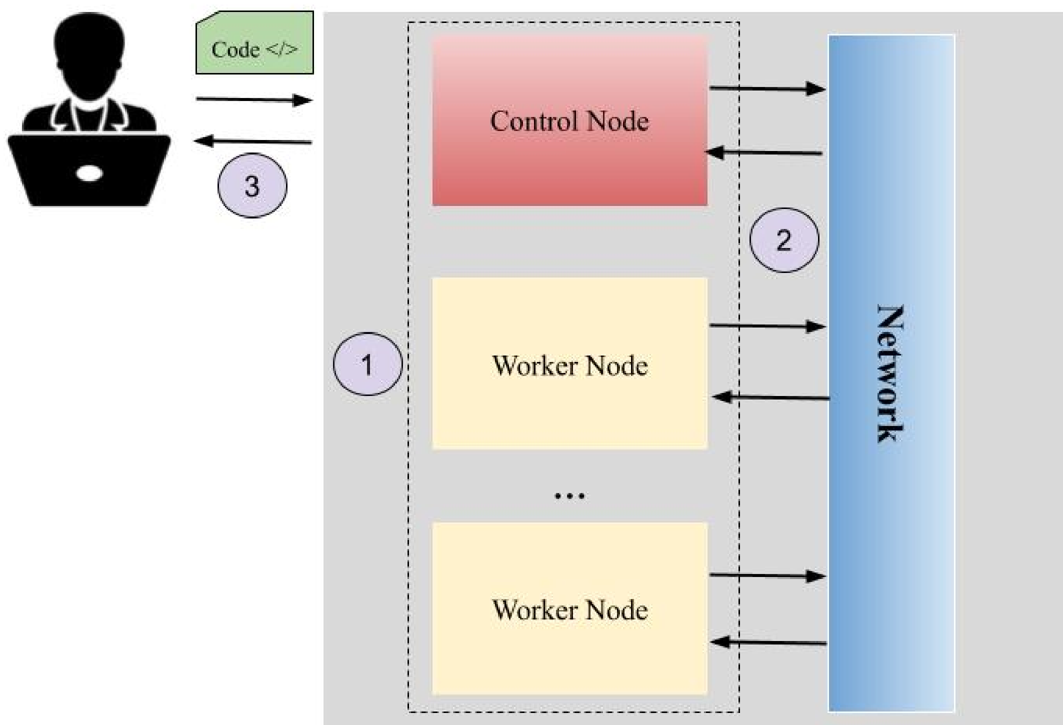
Below, we outline the key contributions of this research:

- **Design and implementation of Escra**: A resource allocation system that operates in an event-driven manner, enabling near real-time allocation of container resources to optimize performance and reduce wastage.

- **Development of THORN-ML, a fault-tolerant RDMA-based network architecture**: The study includes an in-depth evaluation of the system's performance and highlights its impact on training state-of-the-art distributed deep learning models using both micro and macro benchmarks.

- **Design and implementation of smart cloud configuration selection for DDL jobs**: Leveraging a dataset collected from Google Vertex AI, this solution analyzes various deep learning jobs running with different configurations and hardware. It predicts the cost and training time of a deep learning task, offering actionable insights for selecting optimal cloud configurations.

Figure 1.1: Graphical Representation of Contributions in the Thesis: (1) Chapter 2: Escra - Event-driven Subsecond Resource Allocation, (2) Chapter 3: THORN-ML: Transparent Hardware Offloaded Resilient Networks for RDMA based Distributed ML Workloads (3) Chapter 4: Optimizing Cloud Configuration Selection for Cost-Effective Distributed Deep Learning Execution



Chapter 2

Escra: Event-driven Sub-second Container Resource Allocation

First, our attention turns to the critical aspect of resource allocation within the underlying cloud infrastructure when deploying containerized distributed applications such as distributed deep learning. We explore the repercussions of inefficient resource allocation, which can significantly impact application performance and result in additional costs for the cloud service providers.

This chapter 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 resource allocation to manage a collection of containers. Escra performs resource allocation on sub-second intervals within and across hosts, allowing operators to cost-effectively scale resources without performance penalty. We evaluate Escra on two types of containerized applications: microservices and serverless functions. In microservice environments, 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, Escra can reduce CPU reservations by over 2.1x and memory reservations by more than 2x while maintaining similar end-to-end performance.

2.1 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; both result in degraded application performance. When resource limits are set high to prioritize application performance, resources are underutilized which increases deployment cost [85, 75]. Developers pay the cost when cloud providers charge tenants based on resources reserved [147, 3, 9]. Cloud providers pay the cost in cases where developers are charged by usage, such as in serverless computing [6, 8, 15, 29].

Due to this trade-off, setting accurate limits is important. In practice, it is also difficult [67, 139, 147, 87, 42][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, a trend often observed by datacenter operators [170, 120, 102, 87, 97].

Recent work has addressed some of these challenges by leveraging machine learning to predict future needs and then automatically scaling container resource limits based on those predictions [147, 139]. These works eliminate the developer burden of setting resource limits but are

[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 [87].

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 [154, 7, 27, 16]. 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 chapter, 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 resources of multiple containers 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 right-fit allocations to current resource demands and can quickly react to instances of CPU throttling.

Escra 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 that applies resource decisions and reports container usage to the controller. A **Distributed Container** abstraction enforces resource isolation by enforcing per-application resource limits, similar to Resource Quotas found in other container orchestration systems [50, 47, 63, 101]. In these systems, Resource Quotas are enforced at the admission control stage. However, unlike Resource Quotas, a Distributed Container enforces resource limits both at deployment and throughout the lifetime of a container, allowing containers belonging to the same tenant to share compute resources across hosts on the order of milliseconds. Runtime limit enforcement enables Escra to fully utilize the per-application limit even when some containers are using less than their initial deployment allocation. The contributions of our work are as follows:

- We expose fine-grained telemetry data from Linux's Completely Fair Scheduler (CFS) [171]. This allows Escra to quickly track and react to actual resource needs, resulting in both high performance (low latency and high throughput) *and* low cost (minimal slack[2]).

- We implement event-based memory scaling and periodic memory reclamation. Escra uses memory scaling to increase container memory upon an OOM event, rather than allowing the container to be killed. Periodic memory reclamation increases application memory efficiency.

- We show Escra is effective by comparing slack, latency, and throughput performance to recently proposed systems. We reduce application latency by up to 96% while increasing throughput up to 3.2x over a state of the art container orchestrator. These low latency and high throughput rates are achieved while simultaneously reducing the median CPU and memory slack by over 10x and 2.5x, respectively. We show the overhead from the central controller is minimal.

- We show Escra reduces slack and both CPU and memory reservations in serverless applications without increasing application latency, potentially reducing cost to both the developer and the infrastructure provider.

2.2 Related Work

Current container orchestration systems (Kubernetes [32], Borg [173], Mesos [106]) set static container resource allocations. Here we present recent works that instead dynamically scale containers and discuss the limitations of these systems.

Vertical Pod Autoscaler (VPA) VPA is a Kubernetes project that implements automated container scaling through a threshold-based scaling mechanism [101]. VPA sets a target resource

[2] Slack: a container's CPU or memory limit minus its CPU or memory usage

utilization and an upper and lower bound on that utilization. When the container usage hits the upper threshold, VPA scales the container up. When the lower bound is hit, VPA scales the container down. VPA also has the capability to enforce per-application limits via Resource Quotas [50]. A resource quota is a hard resource limit on the aggregate compute usage across all or a subset of deployments or services in a Kubernetes namespace.

Limitations of VPA VPA sets the upper and lower limit scaling bounds far apart. Since scaling a container requires a container restart, VPA only scales a container at most once per minute. The loose scaling-bound limit and infrequent container scaling results in high slack which translates to decreased cost-efficiency.

Autopilot Autopilot is a proprietary Google project that addresses the low cost-efficiency of static container deployments [147]. Autopilot runs a control loop that collects both per-second and five minute aggregated usage data from each container, analyzes it, and then makes a prediction on whether or not a container needs to be scaled. Autopilot uses machine learning predictions to scale container limits as frequently as every five minutes.

Limitations of Autopilot While Autopilot provides an automated mechanism to set limits, it does so at coarse-granularity which causes cost-efficiency and performance issues for two reasons. First, Autopilot's heavy-weight algorithm and periodic control loop prevent it from quickly responding to changes in workloads. As a result, resource predictions are forced to at least match the maximum predicted usage over the next allocation period (Autopilot uses a default 5-minute period). This leads to unnecessary slack. Second, because Autopilot relies solely on prediction, it is unable to correct inaccurate predictions even when resources are available. Inaccurate predictions can cause unnecessary OOMs and CPU throttles.

Firm Firm also uses machine learning to improve containerized application performance and cost-efficiency [139]. While Firm does attempt to minimize CPU reservations, the primary objective of Firm is to reduce service-level objective (SLO) violations. Firm minimizes SLO violations by intelligently multiplexing compute resources to optimize the critical path of an application. Firm is similar to Autopilot because it does not require a pod restart to scale container CPU resources and can update container limits automatically.

Limitations of Firm Firm does not implement seamless or automatic **memory** scaling, requiring users to set static limits. Firm shares the limitations of Autopilot regarding performance and cost-efficiency issues as both frameworks feature a coarse-grained, ML-based feedback loop.

2.3 Introducing Escra

Escra is a container resource allocation system that achieves high performance, cost-efficiency, and strong isolation. Escra automatically scales containers in a fine-grained manner, while providing strong isolation via a new abstraction called a Distributed Container. A Distributed Container allows containers belonging to the same tenant to dynamically share resources across multiple compute nodes while capping the overall aggregate resource usage for a given application or tenant at runtime.

Figure 2.1 shows a high-level view of the four key components in the Escra architecture. The Application Deployer and Container Watcher (①) take a set of YAML files describing a set of Kubernetes deployments, services, and containers. The Application Deployer interfaces with the Kubernetes API to deploy containers. The Container Watcher monitors Escra containers and enables newly deployed containers to start streaming fine-grained telemetry to the Controller. The logically centralized Controller (②) handles the unique, fine-grained telemetry sent from the kernel via kernel hooks on workers (③). These kernel hooks obtain fine-grained scheduler data that is not available in user-space. A centralized controller model can be capable of scaling, as evidenced by production systems for datacenters [62] and geo-distributed network services [57]. The Resource Allocator (④) ingests telemetry from the Controller and makes per-container resource allocation decisions. Finally, similar to Kubernetes's per-node kubelet [32], an Agent is run on each host (⑤). The Agent handles resource updates sent from the Controller and can dynamically scale both CPU and memory container limits without restart on the order of 100s of microseconds. In this section,

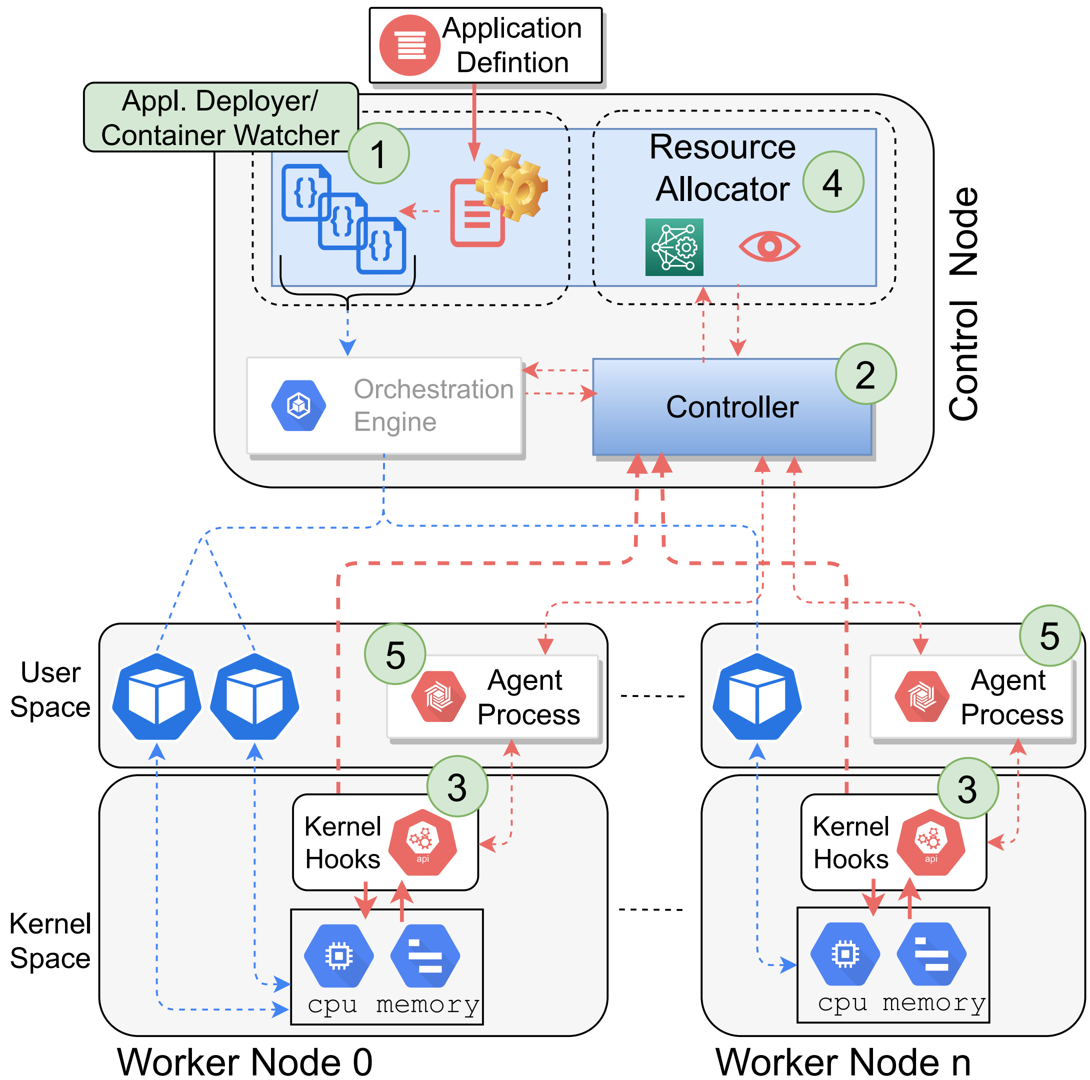


Figure 2.1: Escra Architecture. A single control node manages and controls a set of containers distributed across multiple worker nodes.

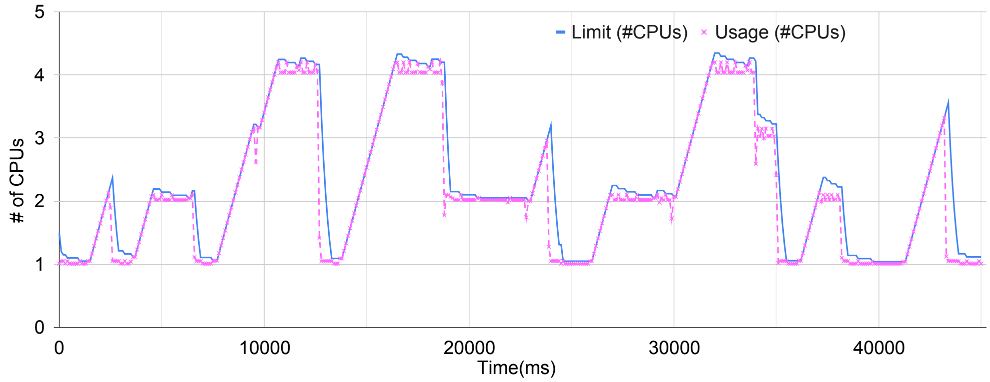


Figure 2.2: Escra's CPU tracking ability under a dynamic workload

we describe Escra's unique ability to make scaling decisions on a fine-grained timescale and in an event driven manner. A complete description of Escra's architecture follows in Section 2.4.

To illustrate the benefits of fine-grained container resource allocation, we deployed and loaded a container with sysbench [112], saturating 1-4 CPUs at any one time. The trace of the application execution with Escra is shown in Figure 2.2. Escra tracks the exact resource needs on a rapid time-scale by reacting to container throttles and OOM events and adjusting resources based on information collected during each CPU scheduling period and at OOM events. The implication of this fine-grained right-sizing is that Escra (1) significantly reduces slack and (2) simultaneously improves performance as applications are being allocated the resources they need rather than being throttled or killed due to OOMs. The remainder of this section provides further insights into how Escra achieves fine-grained resource allocation.

2.3.1 Per-period CPU Telemetry and Dynamic Reallocation

Fine-grained telemetry data is required to minimize slack via fine-grained resource allocation. Our initial analysis of systems that aggregate CPU and memory data (cAdvisor [11], Prometheus [33], Kubectl [32], etc.) found they suffer from inefficiencies stemming from reliance on coarse-grained timescales. Allocating resources quickly is not useful if allocations are based on usage data that is stale or aggregated at insufficient levels. Our goal is to obtain near-instant usage information so Escra never operates on stale data. In order to obtain fine-grained CPU data, Escra uses kernel hooks into Linux's Completely Fair Scheduler (CFS). Upon deployment of each container, the Agent process creates a kernel socket for the container to use to report its metrics to the Controller. To implement fine-grained telemetry, containers report their per-period runtime statistics to the Controller at the end of each period. The telemetry data consists of the cgroup ID of the container, whether the container was throttled in the last period, and the amount of unused runtime in that period.

The Resource Allocator ingests raw container metrics from the Controller and uses two windowed statistics to track unused runtime and the number of throttles. The Resource Allocator uses these statistics to update per-container limits as often as every 100ms. The goal is to proactively update limits in order to keep the container limits just above container usage at all times. We update container CPU quotas using RPCs to the Agent process running on the host of the container, similar to [139].

2.3.2 Reactive Memory Reclamation and Reallocation upon OOM Events

Escra monitors container memory usage and can seamlessly scale memory limits via two custom system calls that hook into Linux's memory cgroup structure.[3] One unique opportunity of fine-grained allocation is the ability to react to OOM events. To achieve this, a kernel hook is added in Linux's memory allocation function, `try_charge()`, to catch a container after it exceeds its memory limit and right before it gets OOMed. This hook combats inaccurate predictions within autoscalers. For example, VPA [101] and Autopilot [147] scale containers at most once a minute and once every five minutes, respectively. There is a chance a container could OOM between allocation decisions. Our kernel hook allows a container to request more memory from the Controller before the container is killed. While this is a reactive mechanism for memory scaling, the request lookup penalty is orders of magnitude faster than a container crash and restart.

One beneficial aspect of this OOM-preventing kernel event is the Resource Allocator can

[3] Docker supports seamless container scaling [19], but Kubernetes does not.

determine how to allocate additional memory resources depending on the state of the node and the application. If there is available memory on the node, the Allocator can simply scale the needy container up. If the node is under memory pressure, the Controller can launch an aggressive memory reclamation process that reclaims memory from other containers on the node with high slack. Not only will this free up memory for the container in need, but it also increases node utilization, reduces slack, and improves cost-efficiency.

2.3.3 Proactive Periodic Memory Reclamation

In order to reduce memory slack, the Escra Controller periodically contacts the Escra Agent on each worker node, asking the Agent to reduce the memory limits of each container on the same node as the Agent. The Agent checks the usage and the limit of each container it manages. If the limit of a container exceeds the usage of the container by more than δ bytes, then the Agent shrinks the container memory limit such that the memory limit minus the memory usage equals δ bytes. Each Agent then reports back the total reclaimed memory from its containers to the Escra Controller. The Resource Allocator can then give the reclaimed memory to other containers experiencing memory pressure.

2.4 Escra Architecture

This section describes the architecture of Escra, our container orchestrator built with Kubernetes, that implements (i) automated container limit settings, (ii) seamless container scaling, (iii) fine-grained resource allocation, and (iv) dynamic, per-tenant resource sharing and collective resource limits enforced at runtime. Escra implements these features using fine-grained telemetry, event-based memory scaling, aggregated application-wide resource limits, and a centralized Controller and Resource Allocator.

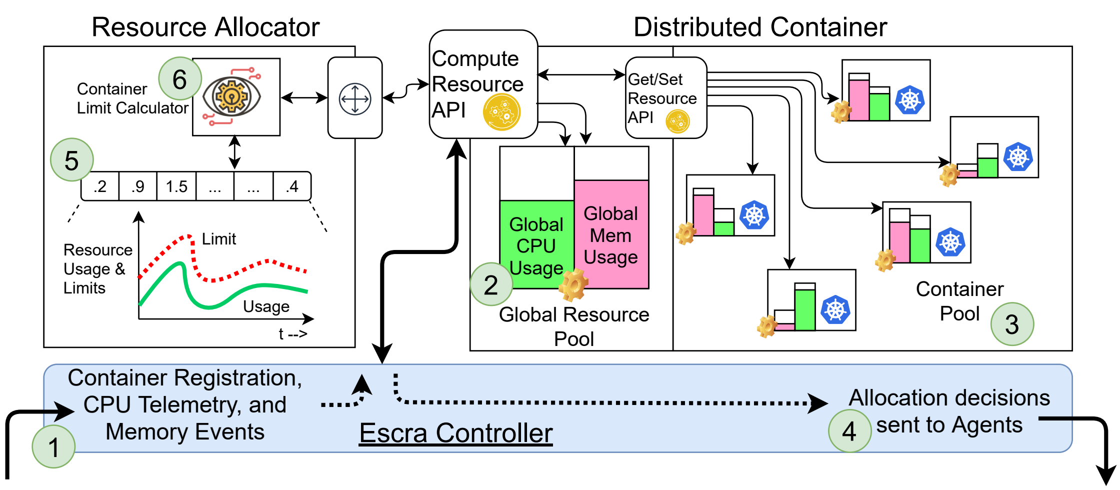


Figure 2.3: Escra Controller, Resource Allocator, and Distributed Container

2.4.1 Application Deployer & Container Watcher

The Application Deployer ingests a Distributed Container configuration as a set of YAML files (Figure 2.1, ①) describing a set of containers, and maximum CPU and memory limits. The maximum CPU and memory limits represent the limit on the aggregate usage of all containers in the application (Figure 2.3, ②). Prior to deploying the containers via Kubernetes, the Deployer sends the global application limits to the Controller. This informs the Resource Allocator (Figure 2.1, ④) of the total maximum usage of the containers in the deployment. Once the Deployer sends the application limits to the Controller, the Controller is ready to accept network connections from each container.

Initial limits are set to bootstrap containers when they first deploy but these limits will be changed by the Controller at runtime. The Deployer initializes the CPU and memory limit of each container to:

$$\frac{global\_cpu\_limit}{\#\, containers} \tag{2.1}$$

$$\frac{global\_mem\_limit * \sigma}{\#\, containers} \tag{2.2}$$

where σ is a configurable parameter representing the percentage of the global application memory limit to be withheld for containers that experience OOM events.

The Container Watcher integrates with Kubernetes to detect container creation. Upon detection, the Watcher notifies the Agent (Figure 2.1, ⑤) located on the same host as the newly created container.

2.4.2 Kernel Hooks

Escra uses kernel hooks to enable fine-grained telemetry and trap OOMs. After an Agent is notified that a new container has deployed, the Agent invokes a custom syscall that carries out three tasks, each implemented via kernel hooks (Figure 2.1, ③). First, the syscall creates a TCP kernel socket to message the Controller (Figure 2.1, ②) and informs the Controller of the existence of the container. The per-container TCP kernel socket will persist for the life of the container. Once the Controller registers the new container, it updates the container's CPU and memory limit based on the global application limits and current application resource use.

Next, the syscall modifies the container's underlying Linux CPU and memory cgroup structures to enable fine-grained telemetry and event handling. For CPU, the syscall hooks into Linux's Completely Fair Scheduler to extract runtime data to stream to the Controller. At the end of each period, the hook writes the container's cgroup quota, unused runtime (the `runtime` variable in the CFS Bandwidth kernel structure), and whether the container was throttled in the last period into a shared FIFO buffer in the kernel[4] .

After the hook finishes writing data to the buffer, the runtime of the cgroup is refilled and the next period begins. Per-container kernel threads consume statistics from the FIFO queue and send the queued CPU statistics over UDP to the Controller. Along with the container quota and remaining runtime, the CPU statistic message also includes a tag letting the Controller know what container the incoming statistic refers to. The hook will report statistics once per-period for the life of the container.

To handle OOM events, the syscall adds a kernel hook in the memory cgroup structure (`mem_cgroup`) for the container. If a container exceeds its memory limit, before it is killed this kernel hook forwards the OOM event to the Controller over the existing TCP kernel socket that was previously used during container initialization. If memory is available in the global application pool, the container can increase its memory limit and continue running.

2.4.3 Controller

The Controller brings all of the system components together and coordinates their interactions. Figure 2.3 shows a more detailed view of the Controller, Resource Allocator, and the Distributed Container abstraction.

[4] Note that per-period unused runtime is not available in userspace and while one could interpret similar data from the `cpuacct` cgroup subsystem, `cpuacct` was never designed for accuracy and was initially designed as a way to showcase the capabilities of cgroups [73].

When containers register themselves with the Controller upon deployment, the Controller creates a logical container object and adds it to a pool of the other Escra containers within the application (Figure 2.3, ②). The logical pool of Escra containers is used to maintain an updated view and status (resource usage, limit, etc.) of the containers it is managing.

Once all containers are deployed and registered with the Controller, the Controller becomes responsible for several additional tasks. The Controller is responsible for launching a periodic memory reclamation process, handling fine-grained telemetry data from all containers, and handling memory requests from containers under memory pressure (Figure 2.3, ①). The Controller is also responsible for carrying out allocation decisions made by the Resource Allocator (Figure 2.3, ④). The Controller is *not* responsible for making those CPU and memory allocation decisions.

The Controller launches a periodic reclamation loop on behalf of the Resource Allocator that triggers each Agent to reclaim excess reserved but unused memory from each container in the cluster. The Resource Allocator determines to what extent each container's memory is resized. Every 5 seconds, the Controller sends a request to each Escra Agent, requesting the Agent to reduce the memory limit of each Escra container, $C(i)$, and send back the amount the container was resized by ψ. This resized value is the amount of memory reclaimed from that specific container. The reclaim process is as follows. The Agent reduces the memory limit on a container if:

$$C(i)_l > C(i)_u + \delta$$

where $C(i)_l$ and $C(i)_u$ are the memory limit and usage of the container, respectively, and δ is a tunable parameter managed and set by the Resource Allocator that represents the memory reclamation "safe margin." If the condition above is satisfied, the container limit is updated via: $C(i)'_l \leftarrow C(i)_u + \delta$, otherwise, the container limit is left unchanged. We empirically set the safe margin to 50 MiB. The amount of reclaimed memory is measured as:

$$\psi \leftarrow C(i)_l - C(i)'_l$$

where $C(i)'_l$ is the resized container limit and ψ is the amount of reclaimed memory. Therefore, for each container that is resized, the Agent passes back to the Controller ψ bytes of memory. The

Escra Controller forwards ψ bytes to the Resource Allocator which then adds ψ bytes of memory into the global memory pool via: $global\_mem\_limit \leftarrow global\_mem\_limit + \psi$. Note that the Controller passes all CPU telemetry data, memory requests, and reclaimed memory updates to the Resource Allocator.

2.4.4 Resource Allocator

The Resource Allocator is the lightweight decision-making component that determines the containers whose resources should be allocated to or reclaimed from. The Resource Allocator is composed of three key components. First, it has a global resource pool for both CPU and memory. For CPU and memory, it keeps track of the maximum application limit (Figure 2.3, ②), the total allocated resources, and the total unallocated (or available) resources (Figure 2.3, ⑥). Second, the Resource Allocator collects fine-grained CPU telemetry data from the Controller and uses a lightweight algorithm to make decisions on whether or not to scale up or scale down individual container CPU limits (Figure 2.3, ⑤). Third, the Resource Allocator consumes *out-of-memory* events sent from the Controller and, based on the globally available memory, increases the memory limit of memory-pressured containers.

If a container is not using up to its allocated resource limit, the Resource Allocator will trigger the Controller to take away those excess resources. However, the Allocator is designed to quickly identify when resources need to be given back to containers and will instruct the Controller to update container limits as needed.

2.4.4.1 Dynamic CPU Allocation

The CPU allocation algorithm consumes CPU telemetry data sent from each container across all nodes in order to share CPU allocations across nodes and remain under the maximum CPU limit (Ω_l). At the end of the container running period t, the Resource Allocator consumes a runtime statistic from the Controller. The runtime statistic for a container i during period t ($C(i)[t]$) includes the container quota ($C(i)_q[t]$) in ms, the amount of unused runtime ($C(i)_q[t] - C(i)_u[t]$) in ms, and whether the container was throttled ($C(i)_{th}[t]$) in the last period t.

The Resource Allocator uses two sliding windowed statistics that track (i) the excess runtime a container has at the end of each period and (ii) if a container was throttled during the last period. Based on these windowed statistics, the Resource Allocator determines whether a container needs or has excess CPU runtime and updates container quotas. A container quota (or limit) during period t is increased if $C(i)_{th}[t] = 1$ and will be increased for the following period $t+1$ via:

$$C(i)_q[t+1] = C(i)_q[t] + \frac{\sum_{t=0}^{n} C(i)_{th}[t]}{n} * \Upsilon(\Omega_l - \sum_{i=0}^{\lambda} C(i)_q[t])$$

where $\frac{\sum_{t=0}^{n} C(i)_{th}[t]}{n}$ is the windowed statistic measuring the average number of throttles over the last n container periods, $\sum_{i=0}^{\lambda} C(i)_q[t]$ is the unallocated CPU runtime for the entire application, λ is the number of containers in the application, and Υ is a tunable parameter that affects the rate at which a container CPU quota is scaled.

A container quota during period t is decreased if $C(i)_q[t] - C(i)_u[t] > \gamma$, where γ is a tunable parameter that adjusts when container quotas should be scaled down. A container quota for period $t+1$ is scaled down via:

$$C(i)_q[t+1] = C(i)_q[t] - \kappa \frac{\sum_{t=0}^{n} (C(i)_q[t] - C(i)_u[t])}{n}$$

where $\frac{\sum_{t=0}^{n} (C(i)_q[t] - C(i)_u[t])}{n}$ is the windowed statistic measuring the average runtime remaining during the last n container periods, and κ is a tunable parameter that affects the rate at which container quotas are scaled down. We empirically found that systems with high variance in CPU usage between periods performed better with a larger Υ and a smaller γ and κ.

2.4.4.2 Dynamic Memory Allocation

This section details the Resource Allocator algorithm for handling *out-of-memory* events received from containers and ensuring the proper sharing of memory resources across an application.

The Resource Allocator determines the amount of additional memory to allocate to containers under memory pressure and the amount of memory to reclaim from containers with unused memory.

The Resource Allocator consumes *out-of-memory* events that are sent from a container just before the container is killed for exceeding its memory limit. Upon receiving an *out-of-memory* event from a container $C(i)$, the Resource Allocator checks if there is unallocated memory available in the global resource pool. If there is no available memory (all global memory has been allocated to containers), the Allocator tells the Controller to reclaim unused memory from other containers in the application (described in Section 2.4.3). We implement *out-of-memory* events in Escra this way to avoid killing a container for exceeding its memory limit when available memory in the application exists.

If the Controller is able to reclaim memory from other containers in the application, the Resource Allocator will allocate a fixed number pages of memory to $C(i)$ by invoking the Agent to update the memory limit of $C(i)$. If the Allocator is unable to reclaim any memory from other containers, $C(i)$ is killed by the operating system (as is standard).

2.4.5 Integrating Escra With Serverless Frameworks

The fine-grained approach to resource allocation in Escra is well suited to serverless environments due to the high degree of multitenancy in serverless systems as well as the short-lived nature of serverless functions. Since functions have short execution times (90% execute in under 1 minute [154]), coarse-grained resource management solutions are insufficient for serverless workloads. Since Escra is fine-grained and designed for use with containers, it is compatible with serverless frameworks that use containers to isolate serverless functions.

We choose OpenWhisk [4], an open-source serverless platform, as an example to illustrate how Escra may be integrated with serverless frameworks. In our configuration, OpenWhisk is deployed via Kubernetes and serverless functions (termed **user actions**) are run in pods. Each pod is deployed as part of the Kubernetes `openwhisk` namespace. Treating OpenWhisk as a single application, one can use the `openwhisk` namespace and invoker `containerPool` memory limit to set global application memory in Escra. We modified pod affinity to ensure OpenWhisk infrastructure was deployed on dedicated infrastructure nodes so there would be no resource contention between architectural components and user actions. While there is no global invoker CPU limit in OpenWhisk, one can set memory and CPU to scale linearly, which indirectly sets a global CPU limit. Escra does not delay container creation in OpenWhisk because the connection between a container and the Controller does not block the container from beginning to execute. Escra already interfaces with Kubernetes so no further modifications are needed for a minimal integration that allows all user action pods to benefit from resource sharing and reclamation.

2.5 Implementation

Escra implementation consists of a total of 14.1k SLOC. The Controller and Resource Allocator are written in C++ and utilize gRPC to communicate with the Deployer, Watcher, and Agents (all written in Go). The Deployer sits on top of Kubernetes and integrates with the Kubernetes deployer API via client-go [14] to deploy Escra containers. Docker is used as the underlying container runtime. The Container Watcher integrates with the Kubernetes work-queue API and communicates with the Agent via gRPC as well.

Escra worker nodes run a custom Linux kernel based on Linux kernel 4.20.16. The custom kernel includes a hook in the CFS cgroup subsystem and in the memory management subsystem. The kernel also includes a custom message structure used for CPU telemetry reporting and memory requests to the Controller. The rest of the kernel modifications include approximately 1,500 SLOC spread across six kernel modules that implement limit resizing and CPU telemetry.

2.6 Evaluation

The goal of Escra is to automatically and seamlessly achieve high performance, cost-efficiency, and isolation. As fine-grained allocation is a key capability of Escra, the first goal of our evaluation is to show how much Escra's highly reactive decision making process is able to improve both performance and cost-efficiency in comparison to common practice (static allocation) and a state-of-the-art system (Autopilot). Our second goal is to show how Escra can reduce the overall reservation requirements for serverless applications, while maintaining application performance; this has the potential to reduce cost for both the application owner and the infrastructure provider.

2.6.1 Experimental Setup

Experiment clusters are created using Cloudlab [91] resources consisting of a control node and worker nodes. Along with the default Kubernetes components, the control node runs the Escra Deployer, Watcher, Controller, and Resource Allocator. Each worker node runs an instance of the Escra Agent.

Microservice Benchmark Applications We first evaluate Escra on a set of four microservice applications running across three worker nodes and one control node. Each node consists of two Intel Xeon Silver 4114 10-core 2.20 GHz CPUs, 192GB of ECC DDR4-2666 memory, and a dual-port Intel X520-DA2 10Gb NIC. We set κ to 0.8, γ to 0.2, and Υ to 20 in the Resource Allocator for all experiments unless otherwise stated.

The microservice applications represent a set of four interactive, real-world benchmarks: (1) *MediaMicroservice* [94] (32 containers): a microservice similar to IMDB [30] where users can search, review, rate, and add films, (2) *HipsterShop* [26] (11 containers): an online shopping microservice consisting of standard browsing and purchasing of various items, (3) *TrainTicket* [60] (68 containers): a microservice that simulates a train ticket booking service consisting of searching, booking, modifying tickets, and (4) *Teastore* [56] (7 containers): a simulated online tea store where users can browse and purchase hundreds of various teas.

For each microservice experiment we load the microservice with one of four workload distributions: a fixed request rate, an exponentially distributed request rate, a bursting request rate, and an Alibaba datacenter trace [2]. The Fixed workload sends requests at a constant 400 requests per second. The Exponential (Exp) workload sends requests in an exponential distribution with $\lambda = 300$. The Burst workload sends a fixed 50 req/sec. with an additional 10 second exponential burst of requests where $\lambda = 600$ every 20 seconds. Finally, the Alibaba workload is sped up by 10x and sends requests at rates anywhere from 56-548 req/sec.

Evaluation Metrics Below is a list of metrics used in this section (derived from [147]):

- **Absolute Slack**: The container CPU or memory limit minus the container CPU or memory usage.
- **Application Throughput**: Measured in successful requests per sec.
- **Application 99.9%ile Latency**: Measured as the 99.9%ile end-to-end latency.

Autopilot Implementation Autopilot [147] is not open-source so we implemented a recreation of the Autopilot ML recommender to compare against Escra. The Autopilot ML recommender is inspired by a multi-armed bandit problem in which an agent tries to use the best set of arms to maximize the total reward gain over time. Some parameters used in the Autopilot algorithm are manually tuned by their engineers (w_o, w_u, etc.). As they did not specify what values they used for these parameters, we tuned them to values that resulted in the best performance.

Note that Autopilot defaults to updating container limits every 5 minutes. We tested the update period of Autopilot at 60, 30, 10, and 1 seconds and saw finer-grained update periods achieve better performance. The throughput of HipsterShop with Autopilot at 1, 10, 30, and 60 second update periods degrades from 422 req/sec. to 382 req/sec. to 279 req/sec. to 108 req/sec., respectively. While we do not know how practical it is to run Autopilot at that granularity at scale, we show comparisons against 1 second intervals as a best case for Autopilot.

2.6.2 Performance - Cost-Efficiency Trade-off

Intuitively, there exists a resource allocation trade-off between performance and cost-efficiency. One can allocate a large amount of resources to eliminate any possible performance penalty (measured in throughput and latency), but this leads to poor cost-efficiency (measured in terms of slack). In contrast, one can significantly under-allocate resources and improve the cost-efficiency, but this is at the price of reduced performance. We further examine this trade-off in the context of

App Comp.	Avg. Δ Latency	Avg. Δ Tput.	Avg. Δ 50% CPU Slack	Avg. Δ 99% CPU Slack	Avg. Δ 50% Mem. Slack	Avg. Δ 99% Mem. Slack
Static vs. Escra	38.0%	25.4%	81.3%	74.2%	55.0%	95.9%
Autopilot vs. Escra	36.1%	54.5%	78.3%	78.6%	26.7%	68.9%

Figure 2.4: Average performance increase and average slack reduction for both CPU and memory between static and Escra and between Autopilot and Escra. Escra improves performance, while significantly reducing slack

both common practice (static allocation) and state-of-the-art (Autopilot), and illustrate that Escra achieves better performance and cost-efficiency than each system, and that the other systems compromised on one of the metrics.

First, we estimated the resources needed for the MediaMicroservice from the Deathstar Benchmark [94] by profiling each container and measuring maximum CPU and memory usage. We then ran the application in underutilized (limits set at 0.75x the profiled max), best-estimate (set at 1.0x), and safe buffer (set at 1.5x) cases. For each case, we measure the end-to-end performance (latency and throughput) and slack (CPU cores allocated minus cores used, and MiBs allocated minus MiBs used). As expected, performance increased (i.e., latency decreased and throughput increased) with more resources allocated; however, slack (resource wastage) also increased. We find the 1.5x allocation level illustrates a sufficient buffer and use that setting for evaluating the trade-offs in comparison to Autopilot and Escra.

For this evaluation, we deployed each microservice and used the workload generation-based benchmarking tool wrk2 [166] with the four different workloads. Each application is evaluated when managed by 1.5x static limits, Autopilot, and Escra. This setup allows us to measure both latency and throughput to quantify the performance in each approach, and slack to quantify the cost-efficiency of each approach. Figure 2.5 shows the resulting *change* in latency and *change* in throughput between Autopilot and Escra and between static limits and Escra for all four applications and workload distributions. Table 2.4 summarizes our results and is broken down in the subsequent sub-sections.

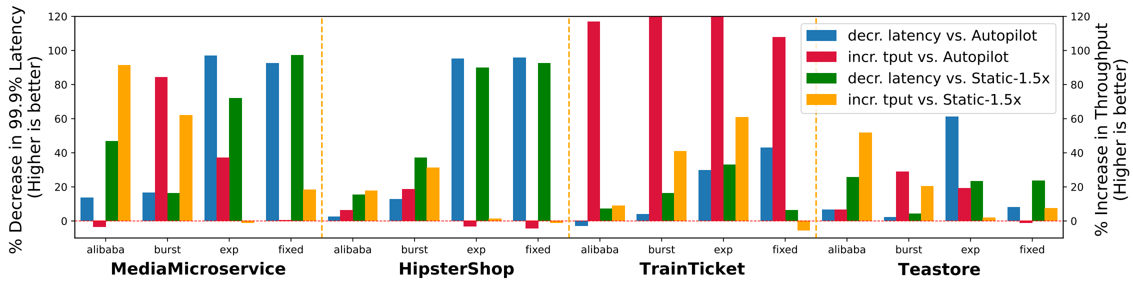


Figure 2.5: Change in 99.9% latency and throughput between Autopilot, the 1.5x measured peak static allocation and Escra. Note: TrainTicket with Burst and Exp workloads experienced a throughput increase of 134% and 324% respectively but are cut off at the top of the figure

2.6.3 Static Allocation vs. Escra

We first look at the change in both latency and throughput between a statically allocated application and an application deployed with Escra. Table 2.4 show that on average, Escra decreases latency by 38% and increases throughput by 25.4% compared to statically allocated applications. Escra can achieve these performance numbers with an average 50%ile and 99%ile CPU slack improvement of 81.3% and 74.2%, respectively. Escra also decreases 50%ile and 99%ile memory slack by 55% and 95.9%, respectively.

In an ideal world, we would not see a performance improvement from Escra over a statically deployed application allocated 1.5 times the peak measured resource usage; the static deployment would never experience any throttles or OOMs. However, this result is a testament to how difficult it is for developers to set resource limits on containers [67, 139, 147, 87, 42]. Not only is it hard to profile containers, since you never know what the workload rate is truly going to be, but also the tools to measure resource usages (especially for CPU) tend to aggregate over seconds to minutes, smoothing out usage spikes [11, 33, 44].

The other reason for the performance difference between Static Allocation and Escra is from the fact that Escra can dynamically share and shift resources between containers at runtime. For example, in a static deployment, when a container is underutilized (C_u) and another container is getting throttled (C_t), C_t cannot use any of C_u's resources. However, in Escra C_u is scaled down while C_t is scaled up (without exceeding the per-application global limit). Escra's ability to shift resources among containers and enforce a per-application limit at runtime, enables an application to fully utilize its allocated CPU and memory. This is a Distributed Container's main difference to Resource Quotas [50, 47]. Resource Quotas are only enforced at container deploy time, so in the case above, C_t cannot scale up because C_u is already deployed and the global limits were enforced on deployment. In the case of VPA [101] (discussed in Section 2.2), the autoscaler would have to constantly kill and restart containers as CPU usages changed.

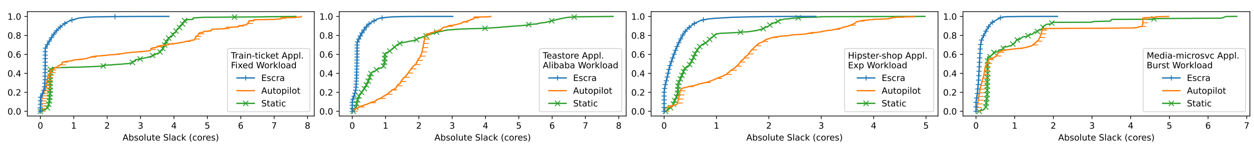
We break down TrainTicket with Fixed and Teastore with Alibaba experiments in the following paragraphs to help illustrate the ability of Escra to achieve both high performance and cost efficiency.

TrainTicket with Fixed Workload Figure 2.5 shows that TrainTicket with Fixed performs slightly worse with Escra than with static allocation, seeing a 5.5% decrease in throughout. Examining the slack in Figures 2.6a and 2.7a, 50% of the time, the static allocation has over 2.5 cores of CPU slack and 256MiB of memory slack. In contrast, Escra has a 50% CPU slack of 0.14 cores (a 17.9x improvement) and memory slack of 49MiB. This experiment shows the trade-off the static deployment makes, sacrificing significant cost-efficiency for a slight performance increase.

Teastore with Alibaba Workload Escra improves latency and throughput of Teastore by 25.7% and 51.6%, respectively. Figures 2.6b and 2.7b show while Escra is able to increase performance, it can do so while reducing 50%ile and 99%ile CPU slack by over 81% and 74% respectively, while also significantly reducing memory slack.

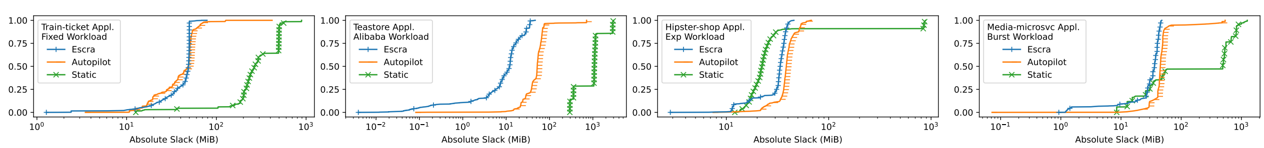
2.6.4 Autopilot vs. Escra

Autopilot aims to reduce slack without sacrificing performance using ML. However, Table 2.4 shows on average, Escra decreases latency by 36.1% and increases throughput by 54.5% compared to Autopilot. Table 2.4 also shows Escra's average 50%ile and 99%ile CPU slack improvement over Autopilot is 78.3% and 78.6%, respectively. Escra also decreases 50%ile and 99%ile memory slack by 26.7% and 68.9%, respectively. We further examine the results of two of these experiments



(a) TrainTicket - Fixed (b) Teastore - Alibaba (c) HipsterShop - Exp (d) MediaMicroservice - Burst

Figure 2.6: CPU slack CDFs comparing Escra, Autopilot, and statically deployed resources across the MediaMicroservice, HipsterShop, TrainTicket, and Teastore microservices with various workloads



(a) TrainTicket - Fixed (b) Teastore - Alibaba (c) HipsterShop - Exp (d) MediaMicroservice - Burst

Figure 2.7: Memory slack CDFs comparing Escra, Autopilot, and statically deployed resources across the MediaMicroservice, HipsterShop, TrainTicket, and Teastore microservices with various workloads. The x-axis is log scale

below to determine how Escra can achieve both high performance and high cost efficiency.

HipsterShop with Exp Workload In a few cases, Autopilot gets some performance improvements over Escra since it trades for performance gains at the cost of slack. Autopilot increases the throughput of HipsterShop compared to Escra by 3.16%. However, Figures 2.6c and 2.7c show Autopilot over allocates resources, with the median slack greater than 1.43 cores and 20% of allocations over 2.38 cores. For Escra, the median slack is 0.12 cores (an 11.6x decrease) with an 80%ile CPU slack of 0.35 cores.

MediaMicroservice with Burst Workload Figure 2.5 shows Autopilot degrades MediaMicroservice with Burst throughput and increases its latency. This indicates that Autopilot fails to quickly react to rapid and significant changes in CPU workloads and memory usages, resulting in low slack but higher latency and lower throughput. For the same application and workload, Escra is able to not only increase latency and throughput performance by 16.6% and 84.3%, but also able to reduce slack over Autopilot. Escra has a 99%ile slack less than 66% of a core and a 99%ile memory slack of 46MiB.

2.6.5 Takeaways

Table 2.4, Figure 2.5, and the four cases above show Escra rarely performs worse than static allocation and Autopilot, but when it does, the performance degradation is small and the slack savings are significant. When Escra outperforms the static allocation and Autopilot, Escra does so with significantly reduced slack, proving that Escra is able to achieve both high performance and high cost efficiency. One of the key reasons for the high performance Escra is that Escra is able to greatly reduce OOMs. In all 32 experiments, Escra experienced zero OOMs, while Autopilot had up to 8 OOMs in a single experiment.

2.6.6 Serverless

This section shows how Escra integrates with OpenWhisk [4] by benchmarking two applications: ImageProcess and GridSearch. We run ImageProcess with one control node, three worker nodes, and two nodes reserved for serverless infrastructure (i.e., OpenWhisk and and a data store). The GridSearch application runs with one additional worker node. Each node is composed of two Xeon E5-2650v2 8-core 2.6 Ghz CPUs, 64GB of DDR-3 memory, and a dual-port Intel X520 10Gb NIC. For both applications OpenWhisk is configured to create each user action pod with 1 vCPU for CPU request and limit, and 256 MiB of memory. We set κ to 0.8 and γ to 0.2 for both applications and Υ to 35 for ImageProcess and 20 for GridSearch in the Resource Allocator.

2.6.6.1 Serverless Benchmark Applications

ImageProcess is a single-function application inspired by the image processing application in [180]. The function reads an image from a database, processes image metadata, creates a thumbnail, and writes the thumbnail to the database. Our workload is simple: an ImageProcess request is sent every 0.8 seconds over 10 minutes. We perform four iterations of the experiment for a total of 3k invocations for each test case. At the beginning of each experiment, we ensure there are no ImageProcess pods running (to ensure initial cold starts).

GridSearch is a traditional approach for tuning hyperparameters in classifiers. This batch-like application [28] uses ~115 serverless function pods to classify an Amazon product review dataset using scikit-learn [54] and tunes the classifier hyperparameters using the GridSearch algorithm. Each function is charged with completing tasks until all 960 tasks are completed. GridSearch uses the Lithops framework [35] for orchestration. We set the Lithops serverless backend to OpenWhisk and the Lithop storage backed to Redis.

The reason Υ is set to different values for GridSearch verses ImageProcess is due to the differences in workload characteristics. In GridSearch, each user action is relatively long-lived as each action is a worker that will complete as many tasks as possible. Thus, it was performant to give Υ for GridSearch the same value used for microservices. In ImageProcess, a user action is a short-lived request. As such, container reuse is common and containers may experience periods of idleness between user actions. Increasing Υ allows containers to more quickly be granted the resources they need as they are created and as they transition from idle (unused) to used (running a user action).

2.6.6.2 Evaluation Metrics

Below are the metrics used in the evaluation of the serverless benchmarks:

- **Aggregate Limits**: Since it is common in serverless systems to bill based on total usage, and serverless providers have a strong incentive to pack as many functions as possible per server, instead of CPU/memory usage per pod we focus on the aggregate of container CPU and memory limits.

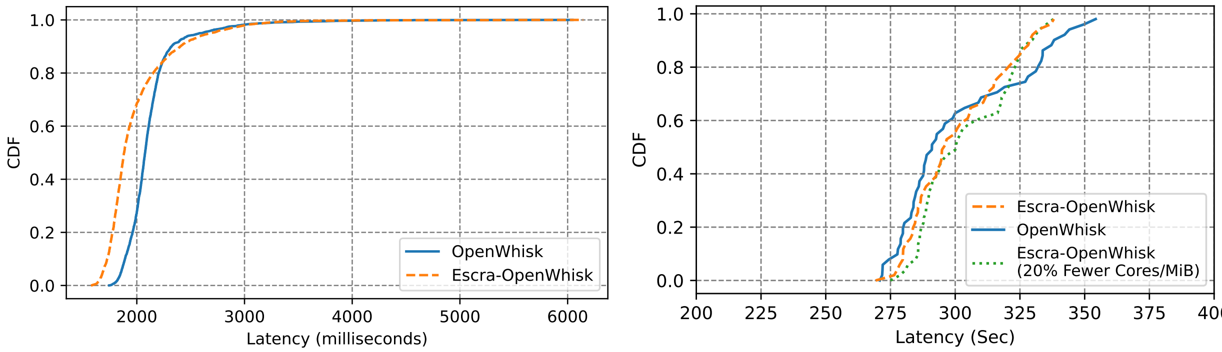
- **Application Latency**: Measured in end-to-end latency per request (ImageProcess) or job (GridSearch)

2.6.7 OpenWhisk vs. Escra + OpenWhisk

2.6.7.1 Performance

We first consider ImageProcess performance for OpenWhisk alone and OpenWhisk + Escra. Figure 2.8a shows that, up to the 80th%ile, OpenWhisk + Escra sees modest performance gains over OpenWhisk alone while the overall 99th%ile latency remains similar for both. The average invocation latency with OpenWhisk + Escra is 1.99 seconds as opposed to 2.12 seconds with OpenWhisk alone. Unlike other applications tested with Escra, ImageProcess requires Escra to handle a variable number of pods as the number of application pods at the start of each benchmark iteration is zero. The similarity in tail latency between OpenWhisk alone and OpenWhisk + Escra indicates that Escra is capable of supporting the dynamic scale-up of application pods needed in serverless environments.

To obtain a CDF of GridSearch application latency, we ran GridSearch on: (1) OpenWhisk alone, (2) OpenWhisk + Escra with the same amount of resources allocated as in the OpenWhisk alone experiment, and (3) OpenWhisk + Escra with 80% of the application resource limits allocated compared to OpenWhisk alone. We ran the application 50 times for each configuration.



(a) ImageProcess request latency

(b) GridSearch application latency

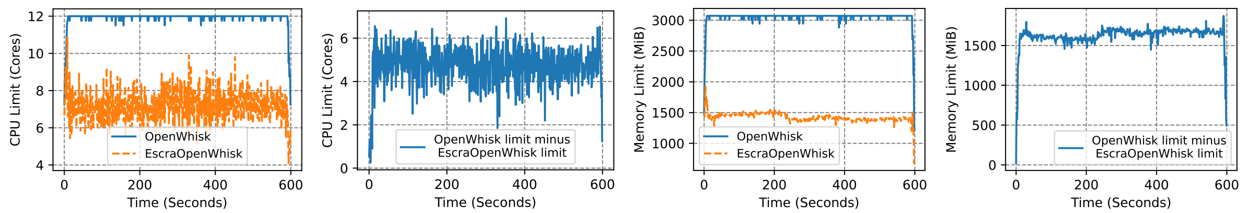
Figure 2.8: Serverless latency CDFs

Interestingly, we observe the same average latency (∼300 seconds) when we run GridSearch by allocating equal resources to OpenWhisk and Escra + OpenWhisk (cases 1 and 2) and only 1% higher average (303 seconds) for case 3, showing Escra can allocate fewer resources to an app and maintain similar performance. As is indicated in Figure 2.8b, Escra + OpenWhisk outperforms OpenWhisk alone at 99%ile and has lower tail latency.

2.6.7.2 Efficiency

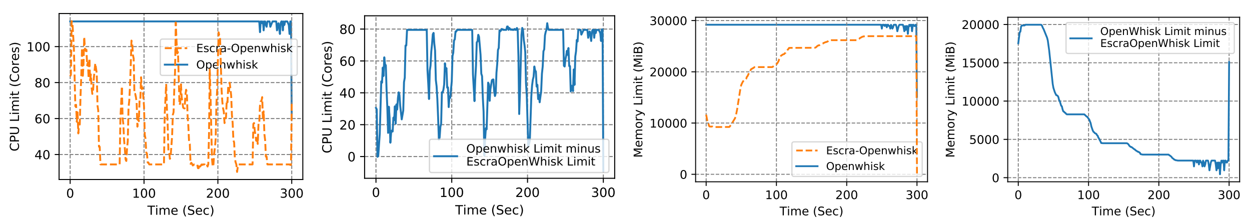
Figure 2.9 shows aggregate CPU and memory limits for OpenWhisk and OpenWhisk + Escra for ImageProcess. On average, OpenWhisk + Escra sets the limit at 7 vCPU whereas OpenWhisk static allocation results in a limit of 12 vCPU, resulting in a savings of approximately 5 vCPU for identical workloads. For memory, the difference in the limit averages around 1550 MiB.

According to Figure 2.10, OpenWhisk allocates 113 vCPUs for GridSearch on average. On the other hand, Escra + OpenWhisk was able to reduce the vCPU allocation to 53 vCPUs. For memory, on average, OpenWhisk sets the application aggregate limit to 29087 MiB while Escra + OpenWhisk is able to run the same GridSearch application with an application limit of 22264 MiB. On average, Escra + OpenWhisk saves 60 vCPUs and roughly 7 GiB of memory space.



(a) Image Process CPU (b) Image Process CPU Savings (c) Image Process Mem (d) Image Process Mem Savings

Figure 2.9: Aggregate memory and CPU limits averaged per second over four test iterations for ImageProcess. We highlight the difference (savings) between OpenWhisk limits and OpenWhisk + Escra limits with the savings graphs.



(a) GridSearch CPU (b) GridSearch CPU Savings (c) GridSearch Mem (d) GridSearch Mem Savings

Figure 2.10: Aggregate memory and CPU limits over 5 minutes of running GridSearch. We highlight the difference (savings) between OpenWhisk limits and OpenWhisk + Escra limits with the savings graphs.

2.6.8 Takeaways

As shown in the ImageProcess and GridSearch benchmarks, Escra only minimally effects function latency while providing significant resource savings on static CPU/memory limits. In sum, Escra increased efficiency while maintaining performance. ImageProcess in particular shows that Escra is able to handle a dynamic and rapid increase in number of application pods. The GridSearch results showcases how Escra can help running batch-like, data intensive, long-running applications with fewer resources but without increasing latency.

2.6.9 Escra MicroBenchmarks and Overheads

2.6.9.1 Why a 100ms Report Period?

Escra uses a 100ms CPU telemetry report frequency for two main reasons. First, 100ms complements the default Linux CFS period. Second, we measured the 99% end-to-end latency performance across various report frequencies every 50ms from 50ms to 200ms. Collecting CPU statistics at the end of every period (100ms) and reporting them directly to the controller resulted in the lowest application latency.

2.6.9.2 Escra Network Overhead

Escra sends usage statistics over UDP to the Controller and the Controller launches RPC calls to the Agent process to update container limits. The peak network overhead measured for 32 containers is 12.06 Mbps. Since the majority of the bandwidth usage comes from the per-container CPU telemetry, we expect the network overhead to scale linearly with the number of containers managed. An investigation into how Escra scales as containers are geographically farther away from the Controller and Resource Allocator (increasing network latency) is left to future work.

2.6.9.3 Escra CPU Overhead

The largest CPU consumers in Escra are the Controller, Resource Allocator, and the kernel threads running on each worker node reporting telemetry data. The Controller consumes the most

CPU out of the three since the the memory reclamation process relies on the cAdvisor API [11], consuming up to 85% of a core. Replacing the cAdvisor functionality with memory limit/usage system calls would greatly reduce the memory reclamation overhead. Without cAdvisor, the Controller and Resource Allocator together use 5.7% of a core with 68 containers. For a cloud-scale analysis, we assume a separate Escra Controller and Resource Allocator that manage each application. Escra Controllers and Allocators are able to manage 1,192 containers per core. Assuming 20 cores per node, a collection of Escra Controllers and Allocators can manage up to 23,859 containers per node. Note, as more containers are registered with the Controller, the mean time between subsequent container stats increases sublinearly.

2.7 Discussion and Future Work

This section discusses how Escra affects cloud ecosystems and describes some directions for future work.

Multi-tenant Building a fully-fledged cluster management system that takes advantage of Escra remains future work. The contribution of this project is that fine-grained, event-driven resource allocation is possible and performs well. While Escra can effectively reduce slack and increase performance, it remains an open question in how such benefits translate to a large-scale, complex, multi-tenant system.

Serverless Our initial implementation of OpenWhisk + Escra is naive in several ways: (1) all containers are treated as the same application; the framework would need to modify this to deploy pods in per-tenant namespaces, and (2) the OpenWhisk invoker remains unaware of the actual CPU and Memory limits being used; it would need to be modified to ingest current usage and limits from Escra. We leave these to future work.

Beyond the efficiency benefits of using Escra in serverless systems, the Distributed Container abstraction may further be useful for billing and accounting in serverless systems [66, 126]. Many commercial frameworks set global limits on serverless applications by setting an invocation limit (i.e., the maximum number of concurrently running functions). With the Distributed Container abstraction, it would be possible to instead limit based on maximum memory or CPU usage. The study of limits and billing using Distributed Containers in serverless systems is a subject of future work.

2.8 Conclusion

This work illustrates how current orchestration systems fail to achieve both high performance and cost efficient container deployments, typically trading performance (throughput, latency) for cost-efficiency (slack) or vice versa. We motivate the need for a fine-grained and seamless container scaling orchestrator and propose a solution: Escra. Escra uses kernel hooks to generate both fine-grained telemetry and OOM handling events that allow a logically-centralized Escra Controller to allocate resources within 100s of milliseconds. As a result, Escra minimizes CPU slack by over 10x compared to our implementation of Autopilot. Escra also reduces application limits in serverless frameworks, saving more than 2x the CPU and memory resources over a standard serverless deployment. Escra's comparison to static approaches, Autopilot, and OpenWhisk deployments indicates fine-grained container scaling finds the balance between performance and efficiency while maintaining isolation.

Chapter 3

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

In this chapter, our focus shifts towards investigating the intricacies of communication across distributed deep learning components, laying the groundwork to emphasize the necessity of an efficient and fault-tolerant network. Such a network stands to significantly enhance the training processes within today's machine learning workloads.

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 chapter introduces THORN-ML, a hardware-offloaded resilient network architecture that is completely transparent to DDL applications and works with commodity hardware. We evaluate THORN-ML on a cluster of 5 nodes with Nvidia A100 GPUs and Mellanox ConnectX-5 NICs, with several 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).

3.1 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 [116].

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 [168], Ray [48], and PyTorch [135], 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 3.2). 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 [100, 133, 92]. 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 [90].). 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.

In this work, we introduce THORN to achieve resilience at the network edge enabling applications to continue making forward progress without having to restart from a checkpoint when failures are encountered at the edge of the network. THORN is designed to fulfill two goals.

Goal 1: applications should not need to be modified. Resilience in the data center network core is transparent to applications; this principle should be applied at the network edge as well. This approach makes the solution more deployable, as there are a variety of widely used applications that could all benefit from a transparent solution. However, achieving transparency poses a challenge due to the wide spread use of RDMA whose programming model is an inherently low-level interface to the underlying NIC hardware.

Goal 2: the hardware (NICs) and protocols (RDMA) should also not be modified. Although there exists a solution that may solve the problem of failures at the network edge[1] [121], this approach requires a custom designed NIC as well as extensions to the RDMA protocol.

The key in THORN to support goal 1 (unmodified applications) is the introduction of a single virtual network device[2] that the application binds to. We then extend the network routing layer in the host, using routing software, to provide failure detection that is aware of the virtualization and available redundancy.

We take advantage of the increasing programmability of commodity NICs (goal 2) using

[1] We say 'may solve', because this solution was designed for dealing with congestion, but in our estimate likely will work to handle failures – though, that was not tested.

[2] A network device in Linux is used to represent, among other things, each individual network port in a NIC—e.g., eth0, eth1.

the Linux switchdev driver by forming Linux **tc** rules that are offloaded to hardware. These rules require a translation layer between the state produced by the routing software (FRR [22], Bird [59], GoBGP [24], etc.) and what the switchdev driver can offload.

Netlink, the protocol to monitor the Linux kernel, does not guarantee delivery of updates; to ensure correctness, we introduce a reconciliation loop.

We evaluate a prototype of THORN on a cluster of five servers, each with an Nvidia A100 GPU (supporting GPU Direct [41]) and a Mellanox ConnectX-5 NIC (dual 100Gbps port), connected through two ToR switches. We show:

(1) THORN supports unmodified applications, tested with GPT-2, ResNet101, and the Nvidia Collective Communication Library (NCCL), (2) there is no noticeable impact in the time to accuracy for applications even with an aggressive failure model, (3) and RDMA traffic performance tests incur a nominal decrease in bandwidth and increase in latency.

Together, the minimal overhead and the ability to eliminate the unnecessary cost of recomputation in the case of edge network failures demonstrates of use and practicality of THORN.

3.2 Motivation

It is natural that faults of a component at the network edge and a fault of the host itself are currently treated similarly as they are not distinguishable. For example, if a cable between the host and the ToR switch gets cut, that host will be detected as unreachable, but the detection mechanism does not know if it was a faulty cable or a downed host. In this chapter we advocate for redundancy at the network edge, such as a NIC with two ports each connected to a different ToR switch. We argue this approach is better than application-level fault handling mechanisms (Section 3.2.1), but hard to support transparently (Section 3.2.2).

3.2.1 Need for Network Edge Resilience

To understand the degree to which fault handling can be improved by resilience at the network edge (as provided with THORN), we explore both the overhead of current fault handling techniques and posit why these overheads are poised to grow.

3.2.1.1 How Host Failure is Handled

Due to the increasing volume of data and the expanding size of deep learning models, DDL has become essential in the current era of machine learning. DDL training is a parallel computing application composed of a number of workers, deployed on a set of nodes, that perform computational tasks and periodically communicating with each other (commonly through collective operations). Modern frameworks employ various forms of parallelism, such as data and model parallelism, to distribute the training workload across clusters of machines.

Training is a single application working towards a single output, i.e., a trained model. With this, the 'global state' of an application is distributed and there is inherent dependence between nodes through the communication that occurs. Failure of one node loses part of that global state. Due to the dependencies between nodes, we cannot just arbitrarily restart a single node; this is a well established problem [115].

Popular DDL frameworks (Pytorch, TensorFlow, Ray) handle node failures through a distributed checkpoint [53, 52, 61]. Developers can configure the frequency of when a checkpoint is taken, typically defined in terms of number of epochs (i.e., iterations where forward and backward propagation occur in neural networks).

When failure is detected, the application must be restarted from the last checkpoint. This applies to all workers, not just the node that failed, because a checkpoint represents the global state for the distributed application.

3.2.1.2 Impact of Failure

To quantify the impact of a host failure, we take measurements on a cluster (fully described in Section 3.5) with 5 hosts, each with a GPU and dual port 100Gbps NIC, connected through two switches, along with a high performance storage server also connected to the switches. There are three components to fault handling: (1) the time to perform a checkpoint – which adds time to each

Table 3.1: Checkpointing and initialization latency/overhead in GPT-2 training on CodeParrot dataset using Megatron-LM and ResNet101 training on CIFAR-10 dataset using Pytorch DDP, along with epoch time for each. The model architecture is annotated as Pipeline Parallel (PP), Data Parallel (DP), and/or Tensor Parallel (TP). Time measurements are in seconds.

Model Arch.	Avg Checkpoint Latency	Model Init. Latency	Avg Epoch Time
ResNet101 (DP)	0.240	5.693	80.0
GPT-2 (DP)	0.771	22.611	4.4
GPT-2 (PP)	0.534	21.283	8.1
GPT-2 (PP+TP)	0.342	20.912	9.9

training iteration, (2) the model initialization time – which is needed on each node when a re-start is needed, and (3) epoch time – which determines the minimum period for taking checkpoints. We measure the effects of failing for two training tasks: GPT-2 (with three different architectures) and ResNet101.

As shown in Table 3.1, each epoch in GPT-2 training using Megatron-LM takes between 4.4 and 9.9 seconds on average for data parallel (DP), tensor-pipeline parallel (PP+TP), and pipeline parallel (PP) configurations. Initialization time after a failure introduces an overhead of 20.9-22.6 seconds for those same three architectures. This results in 27.0-30.8 seconds of unnecessary delay per failure, even when checkpointing after every epoch. For ResNet101 training on the CIFAR-10 dataset using PyTorch Distributed Data Parallel (DP), this overhead can reach upwards of 85 seconds (initialization latency + average epoch time).

These numbers assume checkpointing at every epoch, which is not common in real-world scenarios due to the additional I/O overhead of checkpointing itself and the cost of storing checkpoints. Less frequent checkpoints reduces overheads but exacerbates the overheads of failure by requiring more re-computation for recovery. Additionally, these figures represent the overheads for a 5 node cluster, whereas a larger cluster likely exhibits higher numbers due to increased I/O.

3.2.1.3 Frequency of Failure (is Growing)

It is useful to understand the extent of the costs of edge failures that THORN seeks to address. In a recent paper discussing Llama 3 models, Meta disclosed some operational data on their 16,000 GPU cluster over a 54-day snapshot [90]. Of the 466 job interruptions recorded, 8.4% were due to a network switch or cable failure, and an additional 1.7% were due to NIC failure. While not all interruptions can be attributed to the network edge, we argue that this is still a substantial disruption – especially given the impact (and cost) a failure can have on training.

Increasing model sizes (approaching 10^{26} today, growing at 4.1x per year [140]) and training dataset sizes (at 10^{12} samples, growing at 0.11-0.23 orders of magnitude per year [174]), require larger and larger networks. The large number of components in the system increases the likelihood of failure of individual components while simultaneously increasing the cost of a failure for any idle cycles due to failure.

3.2.2 Challenge with Redundancy

We posit that handling network faults should be transparent to the application, and should not require the development of custom hardware or network protocols. This subsection discusses why the use of RDMA in large-scale DDL poses a challenge to application transparent failure handling.

3.2.2.1 Remote Direct Memory Acess (RDMA) Background

RDMA enables the transfer of data between memory in two different servers without the involvement of the CPU in either server [143, 152]. This can be system memory or the GPU's memory (through GPU Direct [41]). This approach facilitates efficient and effective communication, as CPU cycles are reserved for application processing and the data transfer is not bottlenecked by the CPU. RDMA was originally designed to work over Inifiniband, but is now widely supported over Ethernet fabrics through RDMA over Converged Ethernet (RoCE) as well.

Consider Figure 3.1, which shows the physical components in a system that supports RDMA.

Assume an application running in host 2 has set up a transfer between some memory region in host 1's memory (shown in the figure in red). The NIC has the ability to directly access host 1 memory (over the PCIe bus), which it uses to read memory, formulate a network packet, and send it to the remote system using the assigned queue pair port as the sending port. For host 1, this is RDMA NIC port 1. The host 2 RDMA NIC also has an assigned port for the queue pair (in this case, port 2) to receive the corresponding data. Host 2 will receive the packet through port 2 and continue execution with the data read from host 1's memory. In this scenario, the host 2 processing unit is a GPU.

3.2.2.2 Application Modifications

The programming model of RDMA is inherently tied to hardware as it involves setting up memory regions and context for the transfer (e.g., the destination host). The NIC will create RDMA control messages to establish the context, then the application can use the *libibverbs* API to set up queue pairs. The *libibverbs* API encapsulates the functionality needed to set up the hardware to initiate a transfer. The first calls made by an application are to get a list of devices which support RDMA (`ibv_get_device_list`), and then open one of those devices (`ibv_open_device`). Each port of a NIC appears in Linux as a separate device.

Applications can then set up a production domain (`ibv_alloc_pd()`) and completion queue (`ibv_create_cq()`) for the specific device, then register the memory region to allow the device to read/write data to this memory (`ibv_reg_mr()`) and setup queue pairs (`ibv_create_qp()`). When setting up a queue pair, the application indicates the connection type, and in many cases, including DDL, this will be the 'reliable connection' to enable the hardware to track lost packets and perform re-transmission.

Note that the application is tightly coupled with the underlying device (i.e., specific network port). If a failure did occur, the applicaiton would be required to detect network failures (network awareness), choose an available path (path discovery), and change the device use for the connection (re-initialize the queue pairs and communication channels).

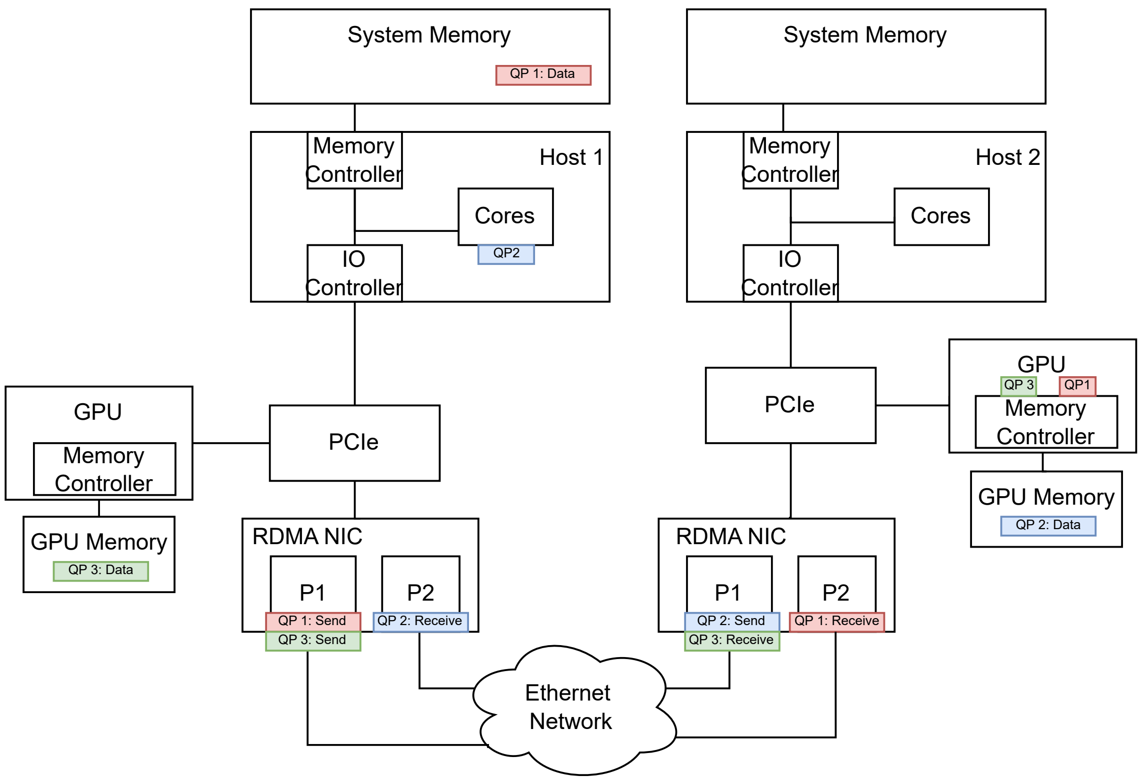


Figure 3.1: Physical view of components in an RDMA transfer.

3.2.2.3 Hardware and Protocol Modifications

Rather than modifying applications, there is a line of work that introduces new hardware and protocols that may support resilience at the network edge. These works are focused on addressing network congestion by proposing alternatives to equal cost multipath (ECMP) as they find that ECMP for ML workloads may lead to congestion [121, 169, 160, 95].

Most of the proposed changes are to network switches, and do not address faults at the network edge [169, 160]. However, in the Multi-Path RDMA (MP-RDMA), the authors propose a custom FPGA NIC to perform network load balancing [121]. The FPGA NIC performs congestion-aware packet distribution to multiple paths using one congestion window for all paths. A specialized path selection algorithm sends traffic on paths with similar delays to reduce out-of-order packets. Fault tolerance is considered between the ToR switch and the spine switch, but not at the edge. MP-RDMA requires extending the RoCE protocol with new fields in the packet header to detect congestion, and requires all nodes to support the new protocol and hardware.

Given that this approach does not consider the components at the edge of the network, it is not suitable (without extension) to solve the need for resilience at the edge. Even if extended, it would face deployment challenges as it requires both custom hardware and a custom protocol.

3.3 Resilient Network Architecture for Unmodified Applications

THORN, illustrated in Figure 3.2, is a host-level network architecture that transparently increases resilience to failures at the network edge. This section describes how THORN provides non-disruptive resilience for DDL applications using RDMA without requiring the application to be modified. This is achieved through 1) creation of a single virtual device that applications can bind to and use with RDMA APIs, 2) tunneling of traffic between these devices on different hosts/workers to hide the underlying network redundancy, and 3) use of standard routing protocols to detect failure and select alternate paths. All steps are designed to be offloaded to hardware to maintain the performance profile of RDMA (discussed further in Section 3.4).

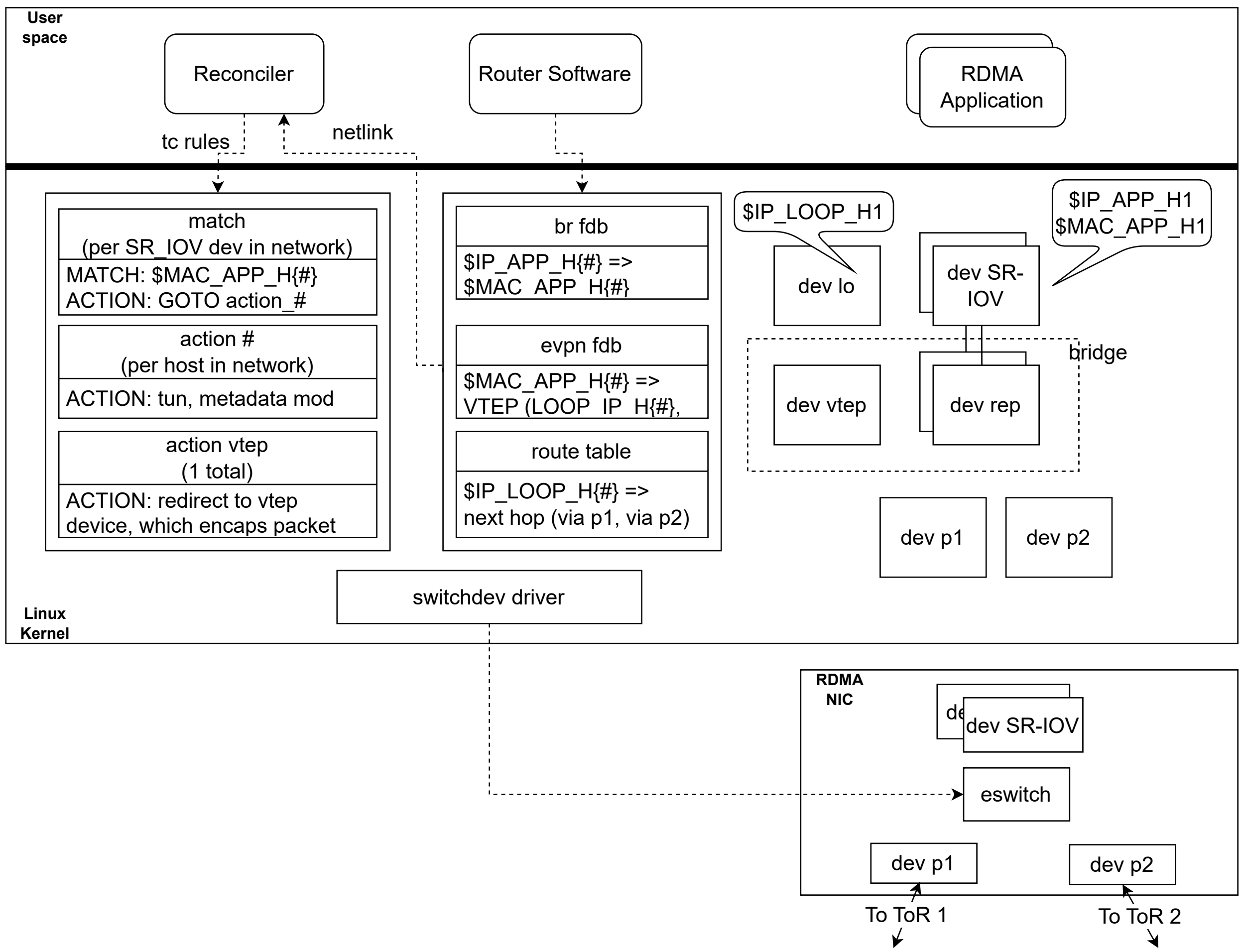


Figure 3.2: THORN Architecture.

3.3.1 Single Virtual Device

As covered in Section 3.2.2, applications are tightly coupled with the underlying device due to the programming model of RDMA. Rather than force applications to change how they interact with the RDMA hardware and become network aware, or change the underlying hardware and introduce extensions to RDMA, we introduce a novel approach which is transparent to applications and works with commodity hardware. THORN creates a single (virtual) device that an application can bind to that can direct traffic to any underlying RDMA port, whether to different ports on the same NIC or to ports on different NICs.

We leverage SR-IOV, an I/O virtualization technology commonly supported in NICs [55]. Setting up an SR-IOV device will result in two devices in Linux, illustrated in Figure 3.2: the SR-IOV device, and a port representor. The SR-IOV device is a physical device that implements the functions of the underlying NIC. The port representor represents a virtual port, and can be interacted with using the standard Linux network stack [38].

The SR-IOV device is assigned both a MAC address and an IP address, which we call MAC_APP_H1 and IP_APP_H1 in Figure 3.2 – IP/MAC to indicate the type of address, APP to indicate that this is what the application sees, and H1 to indicate this is host H1. A DDL application can connect to the equivalent APP addresses for different workers.

As indicated in Figure 3.2, multiple SR-IOV devices may be created. One could be created for each worker running on a host and others created for other application infrastructure, such as connections to storage or databases containing training data; THORN can provide resilience for accessing those services as well.

3.3.2 Bridging to a VXLAN Tunnel

With applications now binding to the SR-IOV devices, THORN must then create network tunnels between SR-IOV devices on different workers in order to hide the underlying network redundancy. We use VXLAN to create the network tunnels, which is commonly used in datacenters

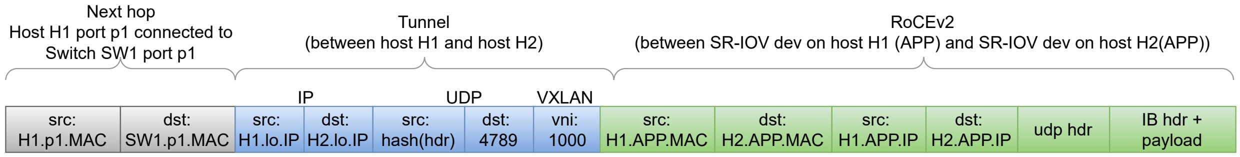


Figure 3.3: Structure of packets as they enter the network from a NIC.

and multi-tenant cloud environments. VXLAN creates a single layer 2 network regardless of the underlying network protocols [122].

VXLAN allows us to create a single virtual tunnel end point (VTEP) device in Linux with a shared VXLAN network identifier (VNI) instead of creating individual tunnels between every pair of end points. In a multi-tenant or multi-application environment, different VNIs can be used to create multiple isolated virtual networks.

THORN must make it appear as if all of the SR-IOV devices for all of the workers (or databases, etc.) are directly attached through a single big switch. To do this, we put all of the SR-IOV devices and the VTEP device into a single Linux bridge, as illustrated in Figure 3.2. The VTEP is used for application-to-application addressing, but a means to support host-to-host addressing (to tunnel the traffic) is also needed. To create this addressing support, we assign an IP address to the loopback device (lo) on each host, which is a device that is always up and can be used to identify the host consistently across all network devices with a single IP address. Once the VTEP is created, we can configure Linux tables to encapsulate and direct traffic. Figure 3.3 shows an example of a complete packet as it would appear when it is sent out of the physical NIC port.

When an application sends traffic, the application knows the IP address of the destination (e.g., some other worker). The bridge forwarding database (`br fdb` in Figure 3.2) holds a mapping between IP addresses (as used by the application, so IP_APP_H{#}) and MAC addresses (as used by the application, MAC_APP_H{#}). The `br fdb` resolves the destination MAC address and forms a RoCEv2 packet, as labeled in Figure 3.3.

The packet is then forwarded to the VTEP device. There, a lookup on the EVPN (Ethernet Virtual Private Network) fdb table using the destination MAC address (MAC_APP_H{#}) provides information about the remote VTEP associated with that MAC address. This includes the IP address of the host with the remote VTEP as well as the VNI of the VTEP. The application packet is then encapsulated (labeled as Tunnel in Figure 3.3). For VXLAN, this includes an IP header (of the source and destination hosts, so IP_LOOP_H{#}), UDP header (which carries a fixed port for the destination), and VXLAN header (which carries the VNI).

Finally, the packet's destination IP address (which is the destination in the tunnel, or LOOP_IP_H{#}) is looked up in the routing table, which contains a list of next hops ordered by best selection. In our example, the packet would be configured to go out device p1. The source is filled in using the MAC address of device p1 and the destination is filled in using the MAC address of the port of the switch connected to device p1.

3.3.3 Scalable Path Selection with EVPN

The previous subsection discussed the various tables and processing steps for handling a packet. This subsection describes how THORN populates and maintains those tables. THORN uses standard routing software on each host, as shown in Figure 3.2. In particular, we use EVPN (Ethernet Virtual Private Network), a standard control plane using the BGP (Border Gateway Protocol) routing protocol. EVPN is used to enable two core functions of THORN: first, it allows each router to advertise the VNIs of the VTEPs it is hosting. This approach provides an automated way for each router to know the IP addresses of all of the hosts with the same VNIs.

The second capability is what makes EVPN highly scalable, at it can be used to exchange the forwarding databases (FDBs). Typically, to determine the MAC address of a destination from an IP address, the Linux kernel issues an address resolution packet (ARP) request which forwards the message to all hosts. This approach is not scalable for larger networks. Instead, with EVPN, BGP is used to exchange this information. As such, anything learned on one host is then known on others, making this highly scalable.

For path selection, BGP peers with neighboring routers. With this peer system, any node knows the available network paths (i.e., it learns about routes available via devices p1 and p2 in Figure 3.2). Then a node can select a best path and set the path into the routing table (the last step discussed in the previous subsection).

When failures occur at the network edge, they are detected in a fast manner through bi-directional forwarding detection (BFD). BFD is a protocol that can rapidly, and with low overhead, detect failures on a link between two connected devices [39, 10]. This protocol is integrated into the routing software. If a NIC fails, a link to the ToR fails, or the ToR itself fails, the routing tables in Linux will be updated quickly. With the routing table in Linux updated, the last step of the packet processing previously discussed only sees one of the devices (between p1 and p2) as a possible next hop, and will choose that as a result.

3.4 Hardware Offload with Unmodified Hardware/Protocols

In Section 3.3, we demonstrated an approach to provide a virtual device that can traverse different network ports (to different ToR switches) with standard Linux-based software. It was designed such that it could be offloaded to hardware with minimal custom software, which we describe here.

3.4.1 Hardware offload with switchdev

With RDMA, local data (whether in system memory or GPU memory) should be directly accessible by the NIC, which can create the needed packet headers and transmit the packets. This means that the tables and configured transformations need to be offloaded to hardware. Without hardware offload, software in Linux needs to craft the packets, which is directly counter to one of the primary benefits of RDMA.

Linux provides a standard driver interface called **switchdev** [21] which offloads the forwarding (data) plane from the kernel. Many NICs support this interface, including the Mellanox Connect-X 5 card used in our experiments.

One challenge is that the switchdev driver does not offload every configuration in Linux – e.g., it does not offload the VTEP device which performs encapsulation in a VXLAN header. However, one aspect that is well supported for offload are rules inserted with the traffic control (tc) utility.

tc is quite powerful at expressing rules following the structure of a match-action pairing, with a rich set of match capabilities and a rich set of action capabilities.

In the case of THORN, the packet starts with an application destination IP and MAC address, and this information needs to determine the associated remote VTEP (IP address of the remote VTEP), encapsulate the packet in a tunnel header, and then forward based on the routing table. To achieve this processing, THORN uses one type of match rule and two types of actions.

tc match: We match on the destination APP MAC address, with one rule per destination SR-IOV device (what the application binds to). Actions can be referenced by an index, allowing multiple matches to share the same action by having each match refer to the action index. This helps with scalability in THORN, as each host may have multiple SR-IOV devices, and each corresponding match conditions can map to the same action index.

tc action 1: This action sets metadata about the tunnel, including the VNI and the LOOP_IP of the destination hosts' loopback device, using the tunnel_key action. One of these actions is required per destination host.

tc action 2: This action redirects the packet to the VTEP device, which will use the metadata to encapsulate the packet in a VXLAN header, and pass the (now encapsulated) packet to be forwarded. Because we only use metadata, only one of these actions is needed per host.

The routing table used for forwarding is automatically offloaded by the switchdev system. This is especially useful since upon failure, the tc matches and actions described above do not need to change, only the routing table which maps destination IP address (i.e., LOOP_IP_H{#}) to a next hop must change. The change to the routing table is able to be handled quickly by the Linux kernel driver.

3.4.2 Correctness despite a best effort protocol

The Reconciler application in THORN is what translates between the kernels network configuration that can't be offloaded and the offloadable tc rules needed to process packets. The Reconciler uses the netlink protocol [37] to monitor the Linux kernel's network configuration. Netlink is a socket family that is widely used for communication between a user space process and the kernel.

Netlink works in both a request-response mode (e.g., tell me the current Linux routing table), as well as a publish-subscribe mode (e.g., update me when the Linux routing table changes). A naive implementation of the Reconciler would subscribe to the tables of interest, wait for an update, write a new tc rule, and repeat. Unfortunately, netlink is a best effort protocol that does not guarantee delivery [37]. For example, it may drop messages due to lack of network socket buffer space in the Linux kernel.

To build a more robust Reconciler, we design an algorithm (shown in Figure 3.4) that ensures we converge on equivalence between the network configuration state as determined by the routing software and the tc rules that can be offloaded. The Reconciler maintains a copy of it's current view of Linux forwarding databases. This view is initialized by requesting a dump from the kernel (a full copy of the state). The reconciler then waits on updates from the kernel. When an update is received, the new state is recorded in the Reconciler internal state. Updates are received by requesting a full dump of the internal state (which mitigates inconsistency due to incomplete updates from netlink). In case of an update that is missed entirely, the reconciler also periodically (based on a configurable timeout) requests a dump from the kernel. This algorithm is effective because it uses the update triggers from the kernel plus the periodic timeout to ensure a missed update is missed for no longer than timeout, but avoids inundating the kernel with dump requests.

3.5 Evaluation

This section evaluates the effectiveness of THORN, using unmodified applications on commodity hardware (Section 3.5.1). We evaluate THORN on three core metrics: (i) The degree to which failures in network edge components cause disruption in THORN. We find that there is no noticeable impact on accuracy over time even with repeated failures over the life of the training. (ii) The impact on bandwidth of THORN for collective communication operations. We find that there is negligible impact in the bandwidth, enabling applications to transfer data at the full rate of the underlying links. (Section 3.5.3) (iii) The impact on latency of THORN for RDMA transfers, dur-

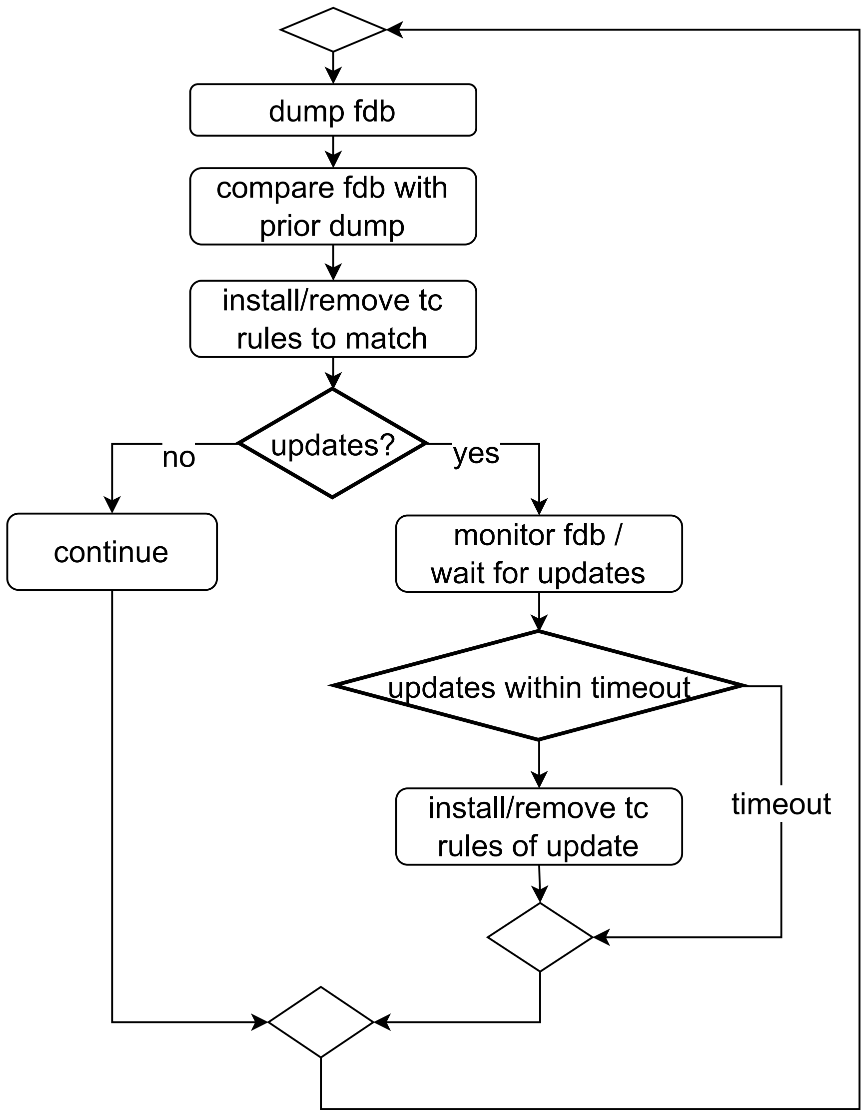


Figure 3.4: Reconciliation loop to ensure consistency between offload rules and Linux forwarding databases.

ing normal operation and during failure. We find that only packets that need to be re-transmitted incur additional latency, and that the detection and fail-over time is so small that it only shows in the 99.9th percentile. (Section 3.5.4)

3.5.1 Experimental Setup

3.5.1.1 Prototype and Testbed

The testbed where we run the THORN prototype is a cluster of 5 hosts and 2 switches. Each host consists of an Intel Xeon Silver 2.10GHz 64-core CPU, an Nvidia A100 GPU with 16 GB of memory, and a dual port, 100Gbps Mellanox ConnectX-5 network card (which supports RDMA). Each switch is a 48-port, 100Gbps Mellanox SN2700. The ports of each ConnectX-5 card are connected to different network switches, and the switches are connected to each other using an active-backup link pair. A 1 TB storage node is connected to each network switch, providing a high-throughput, low-latency log collection and dataset distribution system.

THORN uses Linux 6.1.87, FRR version 10.1 for routing software, a patched version of the rust-netlink library [51](with the modifications being upstreamed), and the Mellanox OFED driver, version 24.01-0.3.3.1. To set up the eswitch on the NIC, we use the Mellanox Firmware Tool (mstflint) [36] along with standard Linux utilities (such as iproute2), totaling 467 lines of scripting. Setting up the VTEPs are standard Linux utilities with 102 lines of scripting. The Reconciler was written with 3679 lines of Rust. We will open source the prototype upon publication.

The testbed consists of only commodity components that are used in datacenters. This setup demonstrates that THORN does not require any changes to hardware or protocols, and is highly deployable and interoperable.

3.5.1.2 Applications

Evaluating with real, unmodified applications allows us to evaluate various modes of parallelism and communication patterns, while also demonstrating that THORN is completely transparent to applications.

ResNet101

The Residual Network (ResNet) is a convolutional neural network designed to enable deep models: ResNet101 [58] is 101 layers deep. We train on the CIFAR10 [13] dataset, a collection of 60,000 32x32 color images that are categorized into 10 classes. We use a data parallel architecture, using the Pytorch Distributed Data Parallel (DDP) API.

CodeParrot GPT-2

To evaluate a complete, large scale application, where we could vary the parallelism architecture for evaluation, we use CodeParrot as an application. CodeParrot [17] is a GPT-2-based language model specifically trained to enable users to generate Python code (in the style of GitHub CoPilot [23]). The authors behind CodeParrot provide datasets of code from GitHub [18] that allows researchers to train the model from scratch.

We conduct the experiments using the Megatron-LM framework [157], which is a research-oriented framework designed for training large-scale language models. It leverages Megatron-Core, a GPU-optimized library, and offers a flexible API for developing custom transformer models. Popular frameworks like HuggingFace Accelerate [1] are built on top of Megatron-LM. We configure CodeParrot with 12 layers and a hidden size of 768. We run the GPT-2 training experiments on the CodeParrot dataset across three different architectures.

Pipeline (model) parallel: The GPT-2 model consists of 12 layers. We used pipeline parallelism where the boundaries between workers is between layers. In our case, we divide the model into four parts to be distributed across four workers, each comprising of three layers.

Pipeline and tensor (model) parallel: We use a hybrid parallel architecture, combining tensor and pipeline parallelism. The model is distributed across two nodes/GPUs horizontally and vertically, requiring a total of four GPUs/nodes for full model-parallel execution.

Data Parallel: We also run GPT-2 training experiments in a data-parallel configuration. In this setting, each GPU maintains a complete copy of the model, while data is distributed in batches across the worker nodes.

NVIDIA Collective Communications Library (NCCL)

We use the NVIDIA Collective Communications Library (NCCL) [127], a specialized library for high-performance GPU communication, as our third application. NCCL offers a comprehensive set of collective operations, including all-reduce, all-gather, reduce, broadcast, reduce-scatter, all-to-all, and point-to-point communication. We test NCCL performance using NCCL Tests [127], a benchmarking tool designed to measure the performance of collective operations across multiple GPUs. NCCL Tests provides insights into the bandwidth achieved during the execution of these operations. Importantly, this setup provides a seamless mechanism for us to experiment with varying communication patterns in different configurations using a library used by many applications.

3.5.2 Impact on Applications Due to Failure

Core to THORN is that a failure of a data center network edge component should not disrupt DDL applications (as current, application-level failure handling techniques do). Here, we measure the actual impact of failures.

3.5.2.1 Accuracy over Time

In this experiment we measure the accuracy in ResNet101 training on CIFAR10 dataset using Pytorch DDP API for 150 epochs. We compare a baseline of training under normal (non-fault) conditions (without THORN) versus one in which network faults are injected and THORN is running and able to handle detection and re-routing.

Faults are injected based on a constant distribution. A fault cycle has a 40 second period, where the device is brought down for 20 seconds, brought back up for 20 seconds, and then the cycle is repeated. We do this during the entire training time for both ResNet101 and GPT-2 training, which means that over the course of training, many faults will occur.

We inject the faults at the network switch, rather than the host, to ensure there is no signal from the operating system of the port going down, and that the failure detection mechanism of THORN is detecting the failure. We selected the port on the device to bring down in a round-robin manner, As shown in Figure 3.5, there is no noticeable difference in achieved accuracy as a function of time, indicating THORN is able to keep the overall training job progressing – even in this extreme fault scenario.

We also compare against a theoretical training accuracy vs time using checkpointing. The first line (labeled Single Fault + Checkpointing), represents a best case if there were only a single fault in the experiment that was subsequently fixed before restarting the training job, but does not include the time to detect and initialize the workers from the checkpoint for the training job. Between 1 and N epochs of training get lost due to the failure, and the line (dotted orange line in Figure 3.5) illustrates the accuracy being stuck at the same amount until the training job re-starts. We assume checkpoints happen every 10 epochs, thus the accuracy gets stuck at the same amount for roughly 13 minutes (based on the measurements from Table 3.1). If multiple faults occur during the training job, as it is the case for the experiment with THORN, the restart delay would occur for every fault. The second line (solid orange line, labeled Many Faults + Checkpointing), represents this case under the extreme fault scenario that we stressed THORN with. Traditional checkpointing alone is not enough to be able to make training progress since there is a fault in every epoch (whereas, THORN is able to continue with similar progress as the baseline without faults).

3.5.2.2 Latency: Across Varying Parallelism Architectures

At a macro level, we do not see a discernible difference in accuracy over time between the case with faults using THORN, and without any faults. We can investigate further by looking at the epoch latency.

This experiment uses the CodeParrot GPT-2 model, which allows us to change the parallelism architecture with simple configuration changes. Our evaluation includes configuration with pipeline, pipeline/tensor, and data parallelism architectures in Megatron.

For this experiment, we measure the time it takes to complete a training iteration. This metric takes into account the failure detection time, failure recovery time, and the associated drop in bandwidth due to the NIC re-transmitting packets as part of the reliably connected RDMA

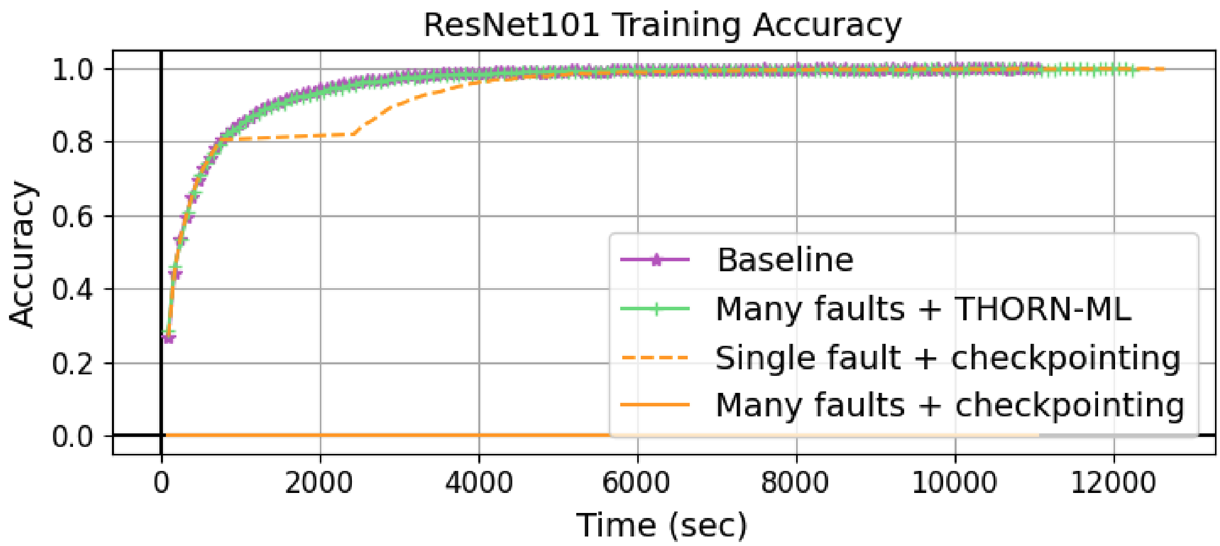


Figure 3.5: ResNet101 training on CIFAR-10 dataset using Pytorch DDP API on 4 nodes demonstrating the time it takes to reach a certain accuracy in normal and fault-injection scenarios with (in green) and without THORN (the rest).

channel. Faults are again injected in the same manner as described in Section 3.5.2.1 (20 seconds down, 20 second up, repeated).

Figure 3.6, shows a cumulative distribution function (CDF) of the time to complete each iteration for the three parallelism architectures (data parallel, pipeline parallel, and pipeline and tensor parallel). In all three configurations, we see little to only modest impact. For data parallel, the added latency at the 90th and 95th percentile is 5.3ms and 7.3ms, respectively, representing a 1.1-1.6% overhead. For the pipeline parallel architecture, the 90th and 95th percentile have an additional 54.9ms and 76.7ms latency, for an overhead of 5.6-7.8%. In the combined pipeline and tensor parallel architecture, the 90th and 95th percentile have an additional 729.4ms and 928.4ms of added latency, representing a 9.2-11.7% overhead.

It should be noted that in all of these experiments with THORN in place, the only overhead in the data plane is an additional VXLAN header, which does not impact performance since it is fully hardware-offloaded.

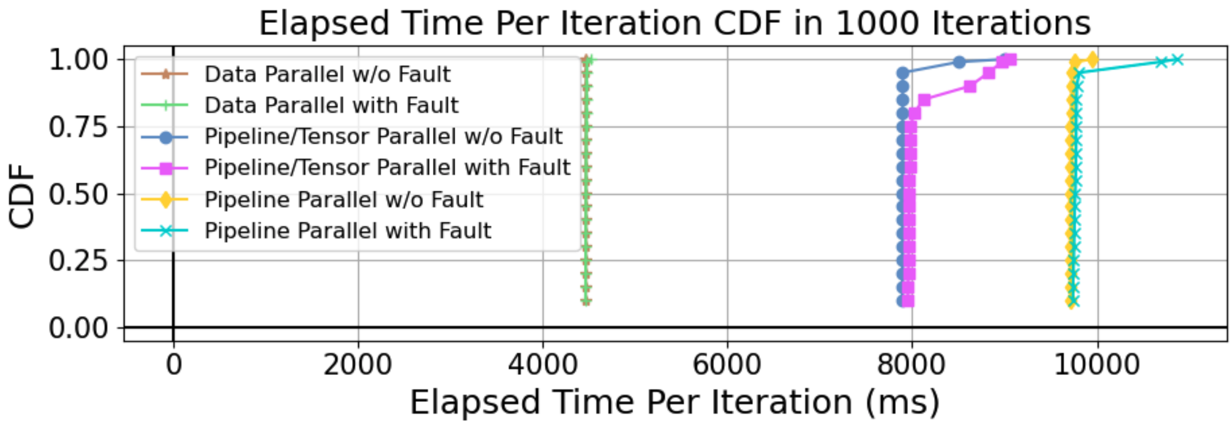


Figure 3.6: Distributed GPT-2 training on CodeParrot dataset performance. Each data point is averaged over 10 iterations.

3.5.3 Bandwidth: Varying Communication Operations

Next, we explore the impact on bandwidth for different communication operations. Here, we used the NCCL test benchmark suite, which provides the means to test several different collective communication operations, while also varying the message size. We ran tests with 100MB, 250MB, 1GB, and 2GB message sizes, all with the same amount of data exchanged ($\approx 32GB$).

Figure 3.7 is the graph for 2GB message size, which is representative of the results for all message sizes. For example, at the 80th percentile, across all operations there is a maximum difference of 0.129Gbps between the fault and non-fault scenarios. We attribute this difference to the known challenge of temporary bandwidth drops upon packet loss in RDMA, which would occur in these cases as the failure is unplanned and, as such, causes packet losses.

3.5.4 Failure Recovery Time

While previous experiments illustrate the impact on the application, here we look into system performance at a finer granularity – in particular, measuring the latency impact for a singular fault.

We leverage an RDMA performance test benchmark tool as it provides better insights into specific performance. To assess the impact of network failures on communication latency, we conduct a series of send transaction experiments between two nodes. Using the *ib_send_lat* command with a message size of 2^{23} bytes, we measure the latency distribution for both normal conditions and scenarios with injected faults occurring midway through the experiment. The resulting graph, shown in Figure 3.8, depicts the cumulative distribution function (CDF) of iteration latency throughout the entire experiment. Note that 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. We see that only packets that need to be re-transmitted incur added latency, and that the detection and fail-over time is so small that it only applies in the 99.9th percentile.

3.6 Related Work

THORN addresses the problem of resilience in distributed deep learning applications. There are two main bodies that we see as most related.

Application Level Fault Handling: The core approach that distributed deep learning frameworks use are around checkpointing [53, 52, 61]. HopLite is one of these checkpointing frameworks [182], which implements collective communication operations atop task-based systems that allow workers to join and leave dynamically. This system improves resilience but does not apply in all scenarios (e.g., applications which cannot be architected with completely independent tasks, and any application using pytorch or Tensorflow). In contrast, THORN can be considered complimentary, as network failures at the edge with HopLite are still disruptive even in these scenarios. Another direction is Ultima [177] which proposes techniques such as adaptive timeouts and transpose-all reduce collectives to reduce the impact of dropped or missed gradients. However, its target is to reduce nominal packet loss (e.g., due to congestion, not failure), and it is unclear how it would work under full failure scenarios (especially a top-of-rack switch failure).

Multi-path RDMA: Recent work by Google reported that the static routing decisions used with RoCE in large scale systems (in this case 4096 nodes) can cause a slowdown of up to 9% and for all-reduce workloads the throughput can be degraded by up to 50% [183]. Meta reported that training jobs had a degradation of more than 30% due to network fragmented traffic and uneven network distributions [95]. These motivated a long line of work in multi-path RDMA, to address the deficiencies of equal cost multipath (ECMP) [121, 169, 95, 160].

The closest work to THORN, which proposes a custom FPGA NIC and extensions to the RDMA protocol to do network load balancing is by Lu **et al.** [121]. This prior work is not transparent to the application and does not work with commodity hardware. Others, either require significant modifications to the applications (whereas THORN is completely transparent) or target the core of the network (which is complementary to THORN).

3.7 Conclusions and Future Work

This chapter presents THORN, an architecture for resilience at the network edge that requires no modification to distributed deep learning applications and can work on commodity hardware and existing popular protocols (RDMA). For application transparency, we introduce a virtual network device that applications can bind to coupled with routing software on the end-hosts to detect failure and select the best routing paths. The proposed system can work with commodity hardware through the support of the switchdev driver, where we bridged the gap between the routing software and the offload capabilities through a reconciliation loop performing rule translation. The end result is a system that enables real applications to proceed without needing to revert to a checkpointed state when encountering a failure at the edge of the network and thus reducing disruption from minutes to milliseconds. Among possible future directions, we will explore integration with the complimentary multi-path RDMA solutions. We will also look to find partners that would enable us to test THORN at larger scales.

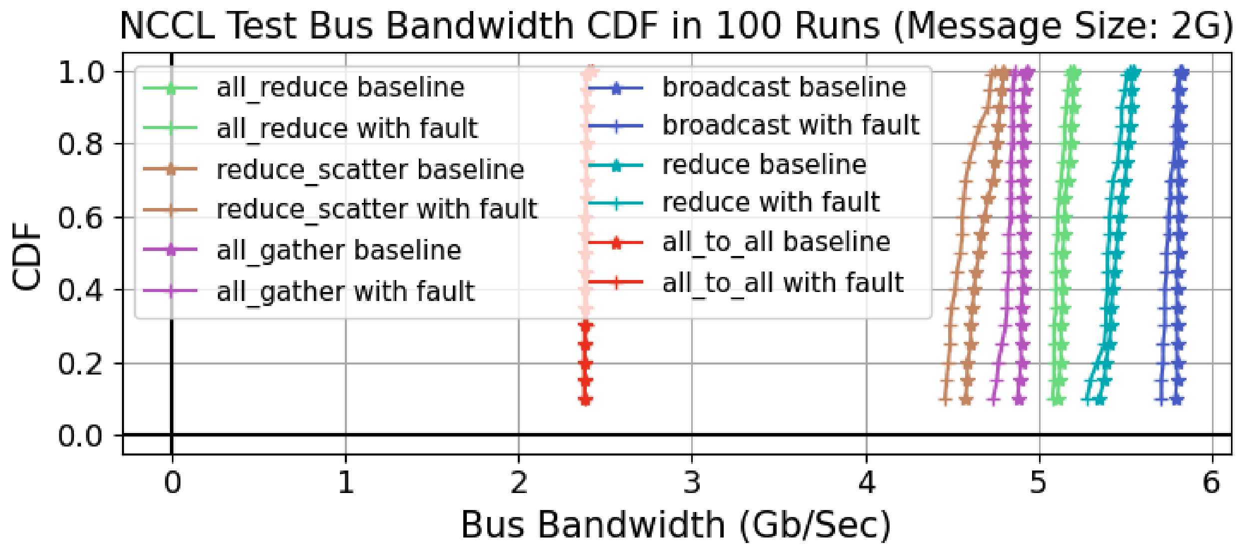


Figure 3.7: NCCL test collective operations performance report with a message size of 2GB for a baseline without THORN and without faults compared to with THORN and with faults. Similar tests were run with 1GB, 256MB, and 1MB, and each showed that THORN handles failures neatly enough to provide nearly the same throughput even in the face of frequent faults.

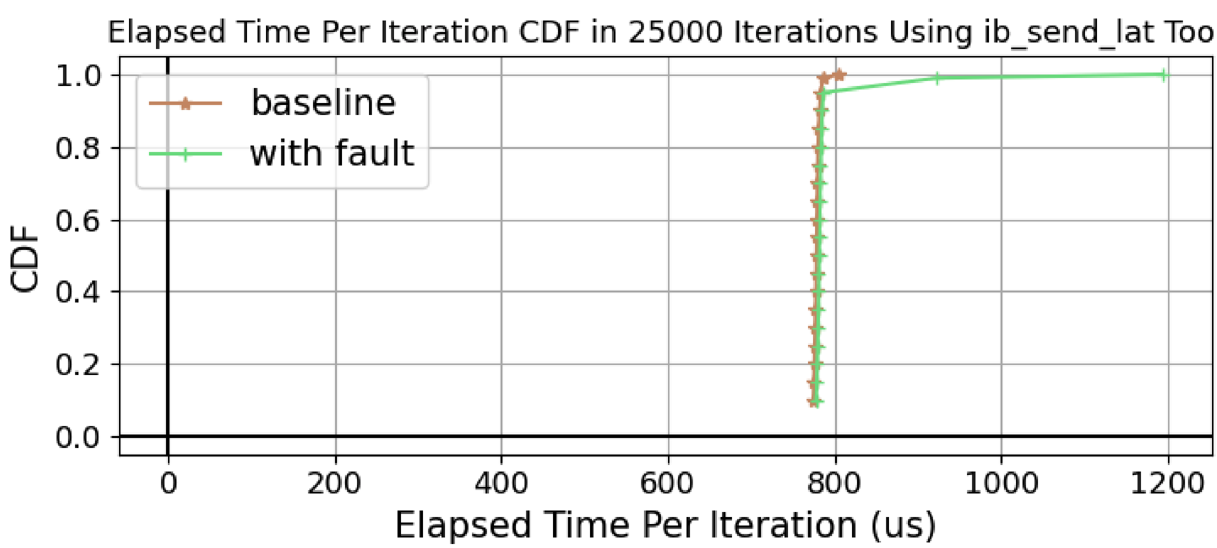


Figure 3.8: Latency CDF using RDMA performance test

Chapter 4

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

In this chapter, we address the critical problem of selecting the optimal cloud configuration for executing distributed deep learning workloads on cloud infrastructure.

The rapid expansion of deep learning models and datasets has outpaced the capabilities of local machines, rendering them inadequate for training due to hardware constraints and prolonged runtimes. Cloud providers offer high-performance infrastructure and AI services, facilitating faster and more efficient model training.

However, selecting an optimal cloud configuration that balances cost and performance remains a significant challenge for developers working with distributed deep learning workloads.

To address this, we propose a predictive framework that leverages a brief, partial run of the workload on local hardware, along with application parameters and cloud specifications, to predict near-optimal configurations in Google Vertex AI. Our approach delivers accurate predictions, enabling developers to make data-driven decisions when selecting cloud resources. For instance, our models predict cost and training time with up to 10% error at the 90th percentile and identify the top 3 cost-efficient configurations out of 126 options for a given training task. By optimizing workloads for cost and performance before deployment, our solution minimizes trial-and-error iterations, reducing costs and improving efficiency.

4.1 Introduction

Deep learning has become ubiquitous, extending its influence far beyond computer science into fields such as medicine, finance, and environmental science, where it drives significant breakthroughs. As a result, an increasing number of non-expert users—individuals who are experts in their domains but lack expertise in computer science—are leveraging deep learning to achieve their goals. This growing demand underscores the need for simplified workflows and accessible tools that enable domain experts to efficiently train models at a reasonable cost and achieve actionable results without requiring deep technical knowledge of distributed systems.

The increasing size of datasets and complexity of deep learning models has led to the rise of distributed deep learning (DDL). DDL enables efficient processing of large-scale datasets and models by distributing the workload across multiple resources. In data parallelism, datasets are partitioned, and each subset is processed by a separate worker node, with the results aggregated to form the final output. This parallel processing significantly reduces training time for large datasets. Similarly, in model parallelism, a model is divided into shards that are distributed across GPUs or machines, allowing them to collaboratively train the model. This approach is particularly advantageous when the model exceeds the memory or computational limits of a single GPU or machine.

Cloud providers such as Google (Vertex AI) [99] and Amazon (SageMaker) [68] offer robust platforms for hosting distributed deep learning workloads, presenting significant advantages over self-managed clusters. These platforms provide access to scalable and powerful infrastructure while abstracting away the complexities of cluster maintenance and fault tolerance. Additionally, they include a rich suite of tools and APIs to streamline the machine learning lifecycle, from dataset preparation to model inference. For example, Vertex AI allows developers to package their training code in a Docker container, upload it to an image registry, and deploy it on a distributed Kubernetes cluster by specifying a few parameters, such as the number of workers and instance types. These services automatically handle cluster provisioning, inter-worker communication, and monitoring,

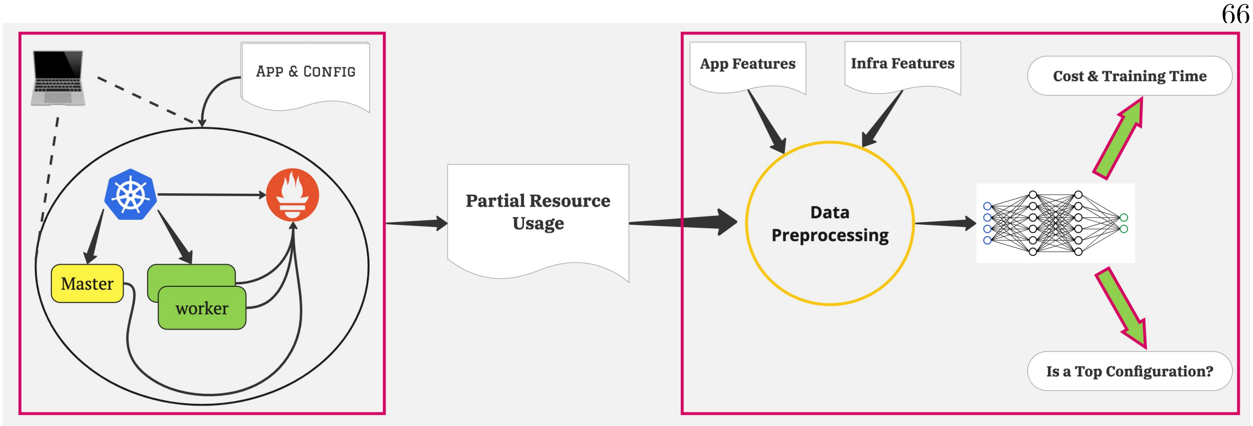


Figure 4.1: An overview of the workflow used for predicting optimal configurations

making them a reliable and efficient solution for DDL applications.

Despite their benefits, cloud platforms pose unique challenges, especially for users with limited expertise in distributed systems. One critical challenge is selecting the optimal cloud configuration to balance two primary objectives: minimizing training time and controlling costs. With hundreds of hardware types and configurations available, and parameters like the number of workers and batch size directly influencing both cost and training duration, optimizing the configuration is non-trivial. Beyond that, even after determining a configuration to use, another challenge is understanding how much a given job would cost and how long it will take to complete. Visibility like this would help engineers make informed decisions about using the cloud resources during their development.

There have been previous attempts at addressing these challenges. One work was Ernest [172], which builds a performance model of an application to then be able to determine the optimal virtual machine configuration to use. While Ernest is relatively low overhead, needing only to run a few samples of the dataset, it is not general. Ernest can only select VM sizes within an instance type family and would need to re-build the performance model again for each instance family type. This is particularly limiting as there are quite a few instance type families per cloud provider, and many cloud providers. Furthermore, it needs re-training for every change in code. For DDL applications, this includes parameters like batch size and number of epochs, which greatly can impact the performance and run time.

Another work is Cherrypick [67], which aims to identify optimal or near-optimal configurations for recurring big data applications by executing sample runs of the full application within the cloud on a given configuration and iteratively refining the configuration. While this generalizes to support a wider set of applications and instance types, it is not scalable. Running distributed deep learning (DDL) tasks multiple times in the cloud before identifying an optimal configuration imposes significant overhead in both time and cost. In CherryPick, a limited search space still required about an hour to find an optimal configuration. As the cloud configuration search space expands, exhaustive sampling is increasingly infeasible.

In this chapter, we introduce a new approach based on developing deep learning models, which is both generalizable and scalable. It builds on two important observations. The first observation is that we can extract resource characteristics of a distributed deep learning training application from a short run in a single container (in our experiments, we ran for 10 minutes). To do this, we can leverage Kubernetes in Docker (KinD) [114], which allows running a DDL application with low overhead, possibly even on a local node, while still enabling us to capture its characteristics such as CPU and memory usage along with its communication patterns. This small data collecting, independent of application parameters, along with the efficiency of inferencing, make this a scalable approach. The second important observation is that insights gained from DDL configuration optimization can be generalized across different applications. Because there is enough commonality between some applications, we can leverage the characteristics of one to aid in the predictive capabilities for another. This reduces the necessity of performing task-specific sampling to derive high-quality configuration candidates, making the solution general.

As illustrated in Figure 4.1, to perform a prediction for a new application, our approach performs a minimal local execution—such as a preliminary run on a laptop or a single-node machine— to extract resource and communication usage features. Combined with DDL task parameters, hyperparameters, and infrastructure characteristics (e.g., available CPU resources, number of workers), we then perform inferencing on pre-trained models that enable predictive insights before deploying the workload in the cloud, specifically on Google Vertex AI.

With this, we build two deep learning models, trained on multiple applications with both a local run, and in-cloud runs with many different configurations (number of workers, machine type, batch size, etc). The first model is built around the assumption that the model will be trained with a data set large enough where the set of applications is representative of any new application for which predictive insights are needed. This is practical for cloud providers, who, through simply running training jobs for their customers, will have a large dataset. This model takes as input the output of the local run, along with application and infrastructure features, and can predict the time and cost. For different application or infrastructure configurations, one can change the inputs, and since inference is fast, this can allow for quick exploration. The second model is built for the cases where a limited data set is available, making precise time and cost estimations infeasible for applications with unseen characteristics. In this model, our method leverages knowledge learned from other training tasks with diverse configurations to identify the most efficient configurations. This allows for effective cost and training time optimization even in the absence of task-specific historical data.

We evaluate our predictive models to demonstrate the effectiveness of our approach, with a collection of 7 applications, run across 630 different configurations. For cost and training time prediction, our models achieved a root mean square error (RMSE) of approximately \$1 and 30 minutes, respectively, accounting for 25% and 27% of the average values on validation sets. This accuracy is limited by our resource limitations, but we fully expect it to be highly accurate with more data. To demonstrate the potential even in scenarios with insufficient data, our top-configuration ranking model in cost optimization was successfully able to identify all optimal configurations during inference. Additionally, by leveraging traditional machine learning models and feature engineering techniques, we conducted a detailed analysis of the collected dataset, uncovering underlying patterns and relationships. This comprehensive evaluation validates our methodology and demonstrates the potential of our approach in enabling cost-effective and efficient cloud-based distributed deep learning.

4.2 Background & Motivation

In this section, we begin with a real-world example to illustrate the motivation behind this research. We then review prior work in the field, highlighting how our approach complements existing efforts in this research direction.

Motivating Example - Consider the following training task: training a VGG-16 model [158] on the EuroSAT dataset [105]. Running this task on a typical laptop is technically feasible, but it would take an unacceptably long time due to the computational limitations of local hardware. To expedite the process, we offload the task to the cloud, choosing Google Vertex AI as the target platform. For this scenario, distributed data-parallel training is appropriate. The training code is implemented using the PyTorch Distributed Data Parallel (DDP) API [135], encapsulated in a Docker container, and pushed to the Google Cloud Container Registry. This container is then deployed using the Vertex AI Custom Training with Custom Containers service [80].

When deploying the task, Vertex AI requires users to configure various parameters, including the type of instances used for training. For simplicity, let us focus solely on the choice of instance type, keeping all other parameters constant. Consider two available instance types: `N1-standard-4` and `E2-highmem-16`. A naive heuristic might suggest selecting the cheaper instance per hour according to the information available on the Vertex AI webpage [81] , in this case, `N1-standard-4`.

However, let us evaluate the actual cost and training time for this task on these two configurations:

- Using `N1-standard-4` with 3 nodes (1 master and 2 workers), the task takes approximately 33 hours to complete, costing $24.29.

- Using `E2-highmem-16` with the same node setup, the task completes in just 7.5 hours. Although the hourly cost of `E2-highmem-16` is three times higher than that of `N1-standard-4`, the total cost is only $19.06.

This example highlights an important insight: choosing the cheapest instance type based solely on hourly pricing can lead to suboptimal outcomes in terms of both cost and time. Conversely, always opting for the most expensive instance type to achieve faster results may lead to unnecessary resource waste. The optimal choice lies in balancing the cost and time trade-offs, a decision that can be challenging without a comprehensive understanding of the interplay between instance characteristics, training workload, and resource requirements.

Background — Prior work has addressed similar challenges in optimizing cloud configurations for computational workloads. For instance, CherryPick [67] employs Bayesian optimization to predict the optimal cloud configuration by leveraging sample runs of a workload within the cloud. Given a search space encompassing all available configurations on Amazon EC2 (66 configurations in their case), it claims to identify the best or near-optimal configuration with high probability after a limited number of trials. However, its effectiveness is primarily suited for recurrent big data workloads, as it requires multiple job executions (at least six, according to their benchmarks) to converge toward an optimal configuration.

Similarly, Ernest [172] constructs performance models for machine learning workloads by having a few sample runs to estimate resource requirements. This approach aims to minimize both resource waste and runtime by predicting the optimal allocation of cloud resources for training tasks.

However, these approaches exhibit fundamental limitations. CherryPick [67], for instance, struggles with scalability as the number of possible configurations and platforms for distributed deep learning (DDL) workloads continues to grow. As acknowledged by its authors, its optimization process slows significantly as the search space expands. More critically, CherryPick relies on executing sample runs within the cloud to guide optimization, introducing substantial overhead in both cost and time. Our experiments indicate that even during the trial phase, training times can extend to several hours per trial, rendering the approach impractical for large-scale or time-sensitive workloads. Furthermore, CherryPick inherently requires cloud-based sample runs for each new task and does not leverage prior knowledge from previous DDL training tasks to make informed predictions for new ones. A more adaptable solution would exploit historical training data to refine predictions over time, reducing the need for costly in-cloud sampling.

Similarly, Ernest [172] presents significant drawbacks. Because it constructs performance models tailored to specific workloads on a given instance type, it necessitates retraining for each new instance type, limiting its adaptability in dynamic cloud environments. Moreover, it is primarily designed for machine learning applications with specific structures and is not explicitly optimized for DDL workloads, which exhibit unique scaling behaviors and performance characteristics that traditional models fail to capture. An effective alternative should leverage the intrinsic properties of DDL tasks to enhance cloud configuration predictions, enabling more accurate and efficient resource selection while minimizing retraining overhead.

In this research, we address these challenges by developing a deep learning-based predictive framework for selecting the optimal cloud configuration, optimized for cost or training time. In addition to extracting features from the application code and DDL infrastructure, our approach utilizes a sample run of the application on a minimal local setup—referred to as the *local run* throughout this chapter—as a representative workload. This local run incurs no additional cost, as it does not execute within the target cloud environment. Our method is inherently scalable, leveraging deep learning inference, and we anticipate that its accuracy will improve as more job records are incorporated into the training dataset over time. Notably, our approach does not require retraining for each new DDL task, as it effectively generalizes across applications by leveraging knowledge gained from prior training tasks.

4.3 Design & Implementation

In this research, we address the limitations of prior works by developing a deep learning-based predictive framework for selecting the optimal cloud configuration, optimized for cost or training time.

Our method is inherently scalable, leveraging deep learning inference, and we anticipate that its accuracy will improve as more job records are incorporated into the training dataset over time. Notably, our approach does not require retraining for each new DDL task, as it effectively generalizes across applications by leveraging knowledge gained from prior training tasks.

This section presents the design and development of a deep learning-based solution for selecting the optimal cloud configuration for distributed data-parallel deep learning (DDL) workloads on Google Vertex AI. Our approach involves training two distinct sets of models, each addressing different aspects of the problem.

In the first set of models (depicted in Figure 4.2), we design a deep learning model to predict cost and training time as continuous values. These models aim to provide accurate estimations, facilitating informed decision-making before executing a training job in the cloud.

In the second set of models (depicted in Figure 4.3), we formulate the problem as a classification task to identify the top configurations in terms of cost or training time for a given workload. By using a binary labeling scheme (0/1), our approach optimizes the selection of the most efficient configurations while ensuring robust generalization to new tasks.

The remainder of this section is structured as follows: First, we describe the feature set used in training our models. We then detail the deep learning model architectures and their design choices, along with the rationale behind our approach and its implications for scalable and accurate cloud configuration prediction.

4.3.1 Feature Engineering

Distributed deep learning (DDL) tasks are governed by a wide range of parameters and hyperparameters at both the application and infrastructure levels, each impacting training time and cost. Table 4.1 summarizes the key features used in our dataset construction and model development.

Application Features: These features are extracted from the application code and are necessary to start the training job. For example, the number of parameters (both trainable and non-trainable) can be obtained using framework APIs such as PyTorch. Hyperparameters like learning rate, batch size, number of epochs, and optimization function are explicitly chosen by the developer. Additionally, features such as the number of layers, input size, and output size can be extracted from the model development code. The *Layers Info* feature represents an array of the

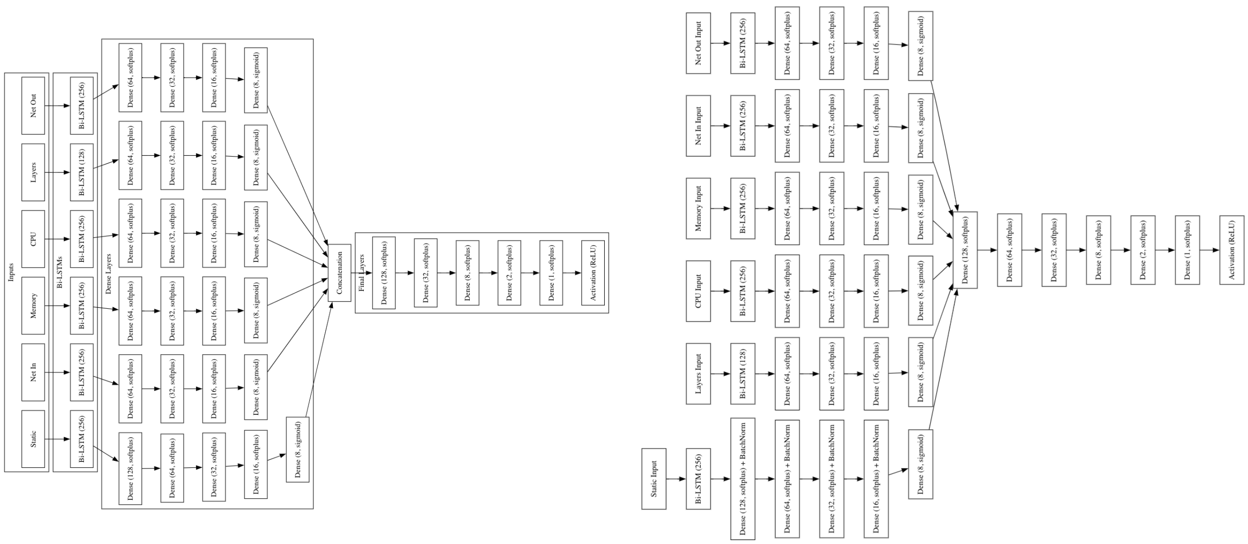


Figure 4.2: Cost (left) and Training Time (right) Prediction Model Architectures for Accurate Prediction.

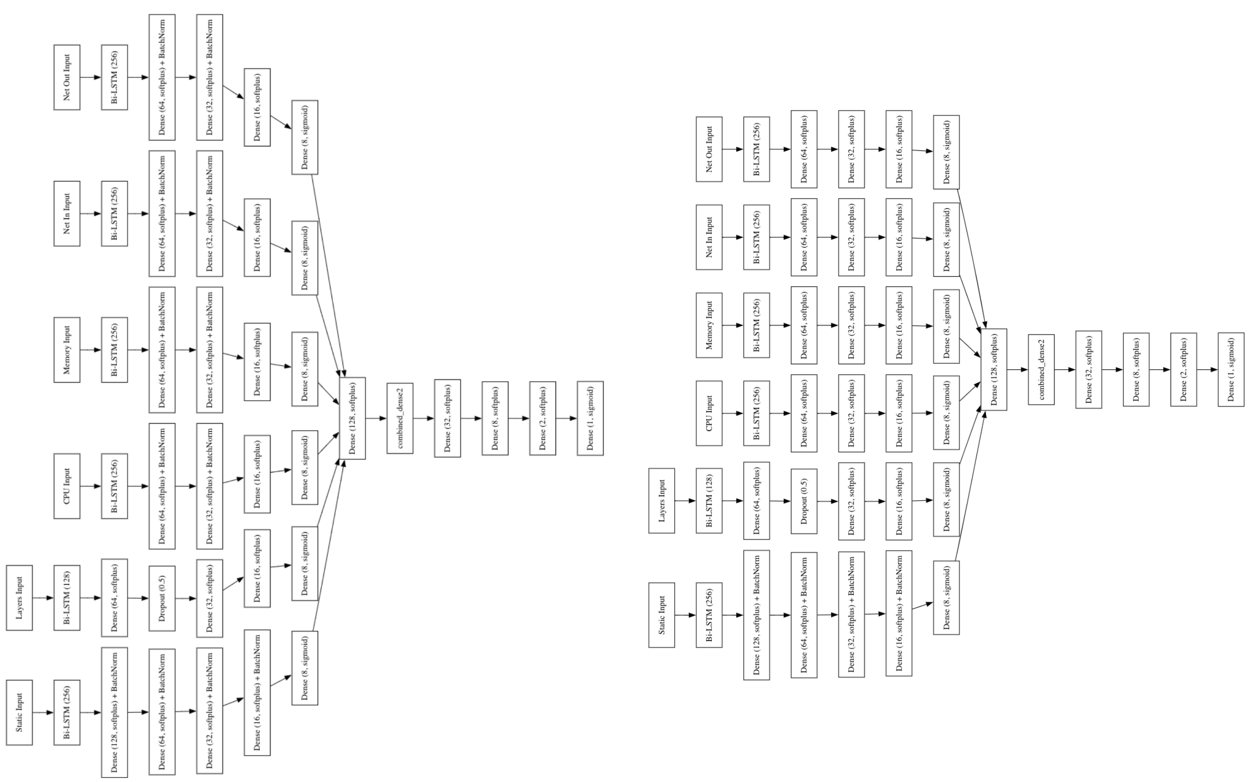


Figure 4.3: Top Candidate Prediction in Terms of Cost (left) and Training Time (right) Model Architectures.

Table 4.1: Features and their categories used in the current model development

Category	**Feature**
Application Features	Number of Model Parameters
	Learning Rate
	Batch Size
	Number of Epochs
	Number of Layers
	Optimization Function
	Input Size
	Output Size
	Layers Info
Infrastructure Features	Number of Workers
	Instance Type
	CPU
	Memory
	Disk
Local Run Timeseries	CPU Usage
	Memory Usage
	Inbound Network Usage
	Outbound Network Usage

specific layers constituting the deep learning model. For instance, layers like `Conv2D`, `ReLU`, and `Dense` are recorded in this array in order. In this group of features, categorical attributes such as layer information and optimizer function were transformed into encoded (e.g. one-hot) vectors to facilitate their use in the training task. Additionally, the input and output sizes were computed as the product of their respective dimensions, providing a single scalar value for each. This approach ensures consistency and numerical compatibility for inclusion in the dataset.

Infrastructure Features: These features represent the target hardware and setup for the training job. Examples include the number of workers, instance type, CPU, memory, and disk size. For a single data point, these features might correspond to values such as 3 workers, `N1-standard-4` instance type, 4 vCPUs, 15 GB memory, and 100 GB disk. Instance type, a categorical feature in this group, is represented using one-hot encoding to facilitate its integration into the model.

Local Run Timeseries: We require a representative workload that can serve as input to our models without incurring additional costs. Therefore, before executing a DDL application in the cloud, we first run the task on a minimal single-node cluster setup to collect resource usage data over a short period. This data forms the *Local Run Timeseries*, as referenced in Table 4.1. The local run captures up to the first 10 minutes of training and records key metrics, including CPU usage, memory usage, inbound network traffic, and outbound network traffic.

By engineering these features, we aim to comprehensively represent both the application-specific and infrastructure-specific factors that influence distributed deep learning tasks. Additionally, the inclusion of local run timeseries data provides a practical and lightweight method to enhance the model's predictive capabilities.

4.3.2 Model Development

The architecture of our deep learning model for predicting cost and training time is designed to effectively leverage the diverse features discussed in the previous section. Our approach begins by grouping the input features into distinct categories: Application/Infrastructure Features, CPU Usage, Memory Usage, and Inbound/Outbound Network Usage. Each of these groups is processed separately through a series of layers to generate an embedding vector, capturing relevant feature representations.

These embeddings are then concatenated and passed through a sequence of dense layers to learn relationships across different feature groups. In the initial processing stage, where each feature group is handled separately, we integrate Bi-Directional LSTMs to capture temporal dependencies and patterns in sequential features, such as timeseries data collected from local runs. Meanwhile, dense layers facilitate the learning of complex interactions between categorical and numerical features. This architectural design ensures that the model can effectively handle the high-dimensional and heterogeneous feature space inherent in cloud configuration selection.

We began with a preliminary architecture consisting of 2–3 layers in both processing stages and iteratively refined the model based on performance metrics, optimizing for predictive accuracy and generalization.

As illustrated in Figure 4.2, the architectures for cost and training time prediction follow a unified design approach, aligning with the reasoning outlined earlier. The final output is produced by a Dense layer with a ReLU activation function, predicting either cost or training time. The model is trained using the Huber loss function, and we optimize for Root Mean Squared Error (RMSE) on the validation set, as this is a regression task.

A key distinction between the two models lies in the training time prediction architecture, incorporates an additional Dense layer before the final output and integrates Batch Normalization layers after Dense layers to refine static input processing. This design choice was guided by hyperparameter tuning, which demonstrated that these modifications improve prediction accuracy by enabling the model to better capture the complex dependencies affecting training time.

For the second set of models depicted in Figure 4.3, designed to classify top-performing configurations for cost and training time, the architecture remains largely similar to those in Figure 4.2. However, for predicting top candidates based on cost, we added batch normalization after each Dense layer in the static feature processing stage, as well as in the CPU, memory, and network time-series processing stages. Batch normalization was also applied to the Dense layers following concatenation to enhance training stability and generalization. For predicting top candidates based on training time, batch normalization was only added after the Dense layers in the static feature processing stage.

Additionally, the final activation function was changed to sigmoid in both models to align with the binary classification objective—determining whether a given configuration belongs to the top-performing set.

4.4 Dataset Collection

To construct our dataset, we executed seven different DDL application in two environments: (1) a local single-node setup to collect local run timeseries data, and (2) Google Vertex AI, where we deployed the applications across various cloud configurations. This process generated approximately 630 data points, where each job is labeled with its corresponding training time and training cost. These data were then unified at the preprocessing stage so that each datapoint has all the feature values listed in Table 4.1.

Applications and the number of datapoints per application is listed in table 4.2. Below, we detail the data collection methodology for each setup.

Table 4.2: Dataset Information for Different Applications

Type	**Training Task**	**# of Data Points**
Text Classification	AG News [161]	126
	IMDB Reviews [163](Custom DNN)	61
	IMDB Reviews [163](BERT + FC)	56
	UDPOS [164]	126
Audio Recognition	YESNO [165]	119
Image Recognition	EuroSAT [162](Custom CNN)	81
	EuroSAT [162](VGG + FC)	63

4.4.1 Local Setup

For collecting local resource usage data, we utilized a single-node environment on CloudLab [91], featuring two Xeon E5-2650v2 processors (8 cores each, 2.6 GHz) and 64 GB of DDR3 memory (8×8 GB RDIMMs, 1.86 GHz).

Distributed deep learning containers were deployed using a single-node Kubernetes cluster created with KinD [114] and managed via the Kubeflow Training Operator [113]. To track container resource usage, we integrated Prometheus [134].

An automation script deployed predefined Docker images to the Kubernetes cluster, monitored resource usage for up to 10 minutes, and saved the data in CSV format. This streamlined the data collection process and allowed for easy dataset expansion.

All DDL applications in this setup ran with three workers and fixed hyperparameters (e.g., batch size, learning rate) to ensure consistency while capturing representative resource consumption metrics.

4.4.2 Google Vertex AI Setup

To systematically explore diverse application and infrastructure configurations, we varied three key parameters: *instance type*, *number of workers*, and *batch size*, as these significantly impact training time and cost. Our experiments covered seven instance types, each evaluated across up to six batch sizes and six worker configurations, resulting in 56 to 126 unique configurations per application.

To automate data collection, we developed a script that accepts job specifications in JSON format and submits them to Vertex AI, enabling parallel execution of multiple configurations. This automation significantly accelerated the data collection process.

Once jobs were completed, training time was extracted from the Vertex AI job metadata, while training cost was retrieved from Google Cloud's billing records via BigQuery. Each job was assigned a unique `job_label` to ensure accurate cost attribution.

This automated approach ensures high accuracy and reliability, forming a robust foundation for our predictive model.

Instance Types and Dataset Scope For this study, we focused on seven instance types: *n1-standard-4*, *n1-standard-8*, *e2-standard-16*, *n1-highcpu-16*, *n1-highcpu-32*, *e2-highmem-8*, and *e2-highmem-16*.

Although our dataset is constrained to these instances and specific configurations, it is designed for extensibility. Future work can incorporate additional instance types, application variations, and hyperparameter settings to further enhance the dataset's scope and generalizability. This scalability underscores the adaptability of our approach for broader DDL workload execution efficiency.

4.5 Evaluation and Data Analysis

This section presents a comprehensive evaluation of our solution on the collected dataset, benchmarking its performance against traditional machine learning approaches such as Random

Forest Regressor. Additionally, we conduct a feature importance analysis to identify strong correlations between features and labels, providing deeper insights into the factors influencing training time and cost, as well as the overall effectiveness of our proposed model.

Our evaluation is structured around two key prediction tasks. In the first task, where we predict cost and training time as continuous values (a regression problem), we focus on minimizing prediction error in the validation set while maximizing the R^2 score to assess model fit. In the second task, where we classify top candidate configurations, we prioritize both precision and recall, evaluating our model using a combination of Area Under the Curve - Receiver Operating Characteristic (AUC-ROC) and confusion matrix analysis.

Finally, we apply off-the-shelf machine learning models as baselines. These traditional models not only highlight the potential improvements offered by deep learning-based solutions but also provide interpretability, helping us uncover strong feature-label correlations and validate the relevance of our selected features.

4.5.1 Deep Learning Models Evaluation

Time and Cost Prediction Model

First, we evaluate the models illustrated in Figure 4.2. For this evaluation, we partition the dataset into training and test sets, ensuring that both contain data points from all seven DDL applications mentioned earlier. This approach assumes that our dataset is sufficiently representative, meaning the test set does not include entirely unseen applications, allowing us to assess model performance under realistic but familiar conditions. In this set of models, as explained earlier, we aim to achieve a low Root Mean Squared Error (RMSE), which serves as a representative metric for prediction error on the validation set. Additionally, we report the improvement in the R^2 score to illustrate how well the model fits our dataset.

We split our dataset into training and validation sets using an 80%-20% split and trained the cost and training time prediction models for 400 epochs. The hyperparameters used in the training process are summarized in Table 4.3. Figure 4.4 illustrates the decrease in the loss metric for both the training and validation sets over the course of training.

Table 4.3: Hyperparameters for the Cost and Training Time Prediction Models

Hyperparameter	Cost Prediction Model	Training Time Prediction Model
Loss Function	Huber (delta=1.0)	Poisson
Learning Rate	0.0001	0.0008
Optimizer	Adam	Adam
Batch Size	32	32

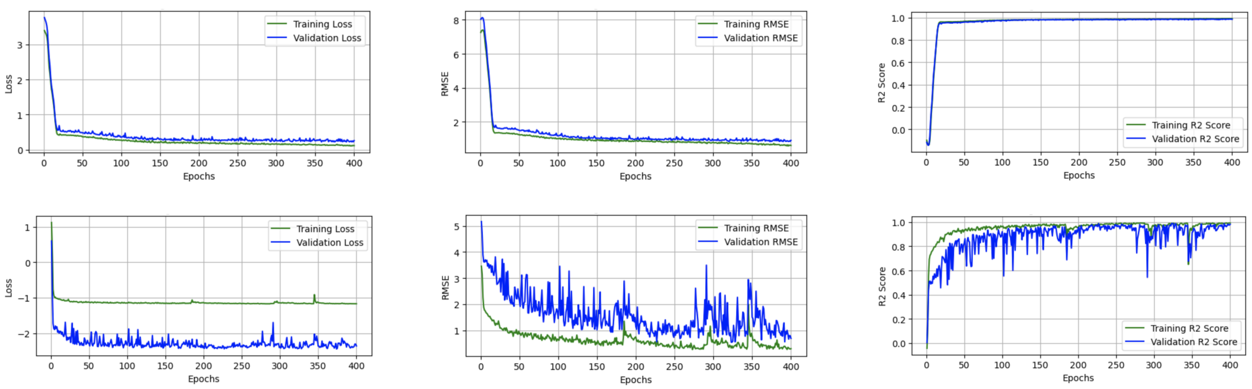


Figure 4.4: Cost (top row) and Training Time (bottom row) Prediction Models Training Performance. Loss values, R^2 score, and RMSE metrics are captured.

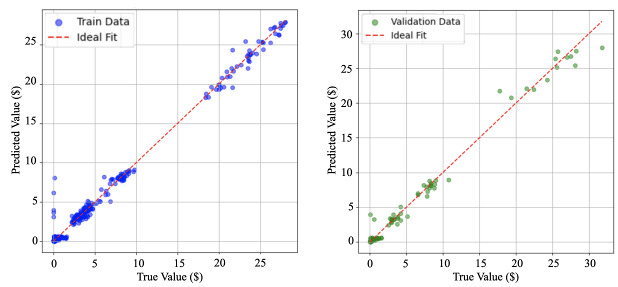
Key Observations: The loss metrics for both models consistently decreased over time, with no indications of overfitting or underfitting, demonstrating strong generalization to unseen data.

For the cost prediction model, the RMSE reached approximately 0.88 on the training set and 1.07 on the validation set. Given that the average cost in these sets was $3.8 and $4.1, respectively, the RMSE corresponds to 23% and 25% of the average cost. Similarly, the training time prediction model achieved RMSE values of approximately 0.33 and 0.53, with corresponding average training times of 1.6 and 1.96 hours. This translates to RMSE values of 20% and 27% relative to the average training time.

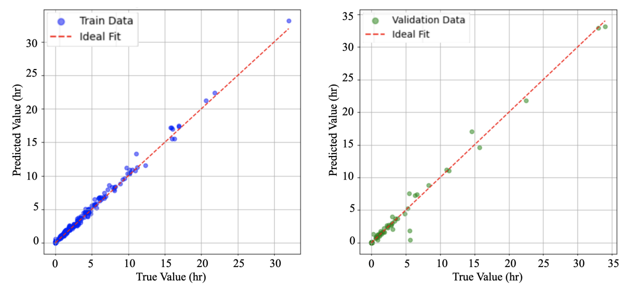
Moreover, the R^2 scores consistently improved throughout training. By the end, the cost prediction model achieved an R^2 score of 0.98 for both the training and validation sets, while the training time prediction model reached 0.99 for both. These high R^2 values indicate that the models capture nearly all the variance in the data, reinforcing their reliability and accuracy.

Key Takeaways: An RMSE between 20% and 30% of the average value in continuous predictions indicates a strong fit, demonstrating the reliability of both models. Furthermore, R^2 scores approaching 1 confirm that the models effectively capture the underlying variance in the data and learn meaningful patterns, reinforcing their predictive accuracy.

The confusion plots, depicted in Figure 4.5, show that true versus predicted values are predominantly centered around the $y = x$ line, further validating the models' predictive capabilities.



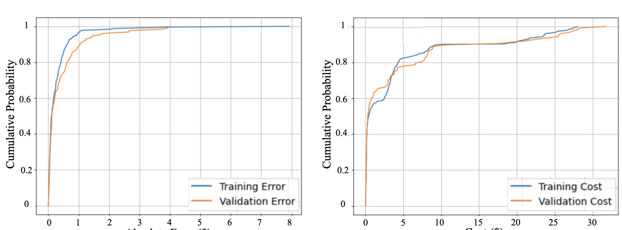
(a) Cost ($) Model Confusion Plots



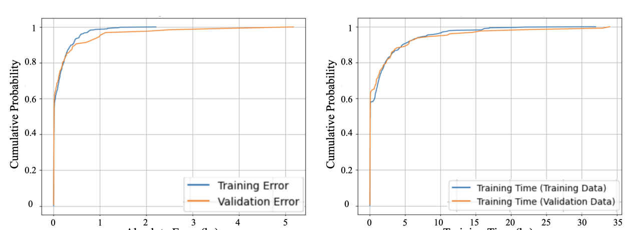
(b) Training Time (hr) Model Confusion Plots

Figure 4.5: Confusion Plots for Cost and Training Time Prediction Deep Learning Models. Line $y = x$ shows the ideal fit.

To gain deeper insights into the prediction performance, Figure 4.6 presents the error CDF graph alongside the CDFs of cost and training time data.



(a) Cost Value and Cost Prediction Error CDF Distribution



(b) Training Time Value and Training Time Prediction Error CDF Distribution

Figure 4.6: Cost and Training Time groundtruth values alongside their prediction error CDF distributions.

Key Takeaways: The CDF for cost shows that 90% of the cost values lie below $9.72 and

\$13.56 for the training and validation sets, respectively. For training time, the 90% percentile values are 4.65 and 5.37 hours. Comparatively, the error CDF indicates that 90% of the cost prediction errors are within \$0.68 and \$1.36, and 90% of the training time prediction errors are within 0.39 and 0.46 hours for training and validation sets, respectively. These findings highlight that the errors are small relative to the actual data distribution, demonstrating the reliability of the prediction models.

Top Configurations Classification This approach simulates a scenario where the training set lacks a fully representative set of applications, such that the test set contains a DDL task with unseen characteristics. While this setup may lead to less accurate absolute predictions of training time and cost, it prioritizes maintaining relative accuracy, enabling the identification of top configurations in terms of cost and training time. To evaluate the model's ability to generalize to previously unseen training tasks, we adopted a different train-test split strategy. Specifically, we excluded all datapoints associated with a particular training application—AG News [161], consisting of 126 datapoints—from the training set and reserved them exclusively for validation. The key metrics in this evaluation are the model's ability to accurately identify the top configurations among the 126 datapoints (i.e., correctly assigning a label of 1) and its effectiveness in minimizing false positives.

For training, we employed a binary cross-entropy loss function and optimized the model using the Adam optimizer with a learning rate of 0.0001.

To construct the classification labels, we ranked all configurations based on both cost and training time. A configuration was assigned a positive label (1) if it satisfied two conditions: (1) it ranked among the top five configurations in terms of lowest cost or fastest training time, and (2) its cost or training time remained within 1.5× of the absolute minimum, ensuring that only configurations with no more than a 50% increase over the optimal value were considered viable candidates.

Key Takeaways: Figure 4.7 presents the AUC/ROC curve, demonstrating the model's ability to identify top-performing configurations. Given the inherent class imbalance—where only a small subset of configurations qualify as optimal—AUC serves as the most reliable performance metric. The model exhibits strong predictive capability, achieving an AUC exceeding 0.95. These results indicate a high potential for practical deployment, with further improvements expected as larger-scale training data becomes available.

When evaluating cost-based top configuration classification, we set an appropriate threshold to distinguish between optimal (1) and suboptimal (0) configurations. The model correctly classified 3 configurations as top candidates from the 126 AG News datapoints, with no false positives. Notably, the ground truth contained only 5 top configurations, meaning the model effectively reduced the candidate search space by over 97%, narrowing down from 126 possibilities to only 3 viable options without requiring any prior observations or trial executions.

For training time-based classification, an optimized threshold resulted in 21 configurations being classified as top candidates. Among these, 4 were true positives, while 17 were false positives. Given that the ground truth contained 5 optimal configurations, the model reduced the search space by over 83%, significantly streamlining the selection process. Despite some false positives, this outcome is promising, as it enables developers to prioritize a small subset of configurations without incurring additional computational costs or conducting exhaustive empirical evaluations.

These findings underscore the model's utility in guiding configuration selection for training tasks on Google Vertex AI, substantially mitigating the need for costly and time-intensive trial runs. By leveraging this approach, developers can make data-driven decisions with high confidence, leading to improved efficiency in cloud-based machine learning workflows.

4.5.2 Machine Learning Models Evaluation

This section establishes a baseline using traditional machine learning models and demonstrates that our deep learning-based solution outperforms them, even if marginally. We anticipate that as the dataset grows, the search space expands, and feature non-linearity increases, the performance gap will widen, making traditional models less effective. Additionally, the interpretability of machine learning models provides valuable insights into feature-label correlations, which can inform

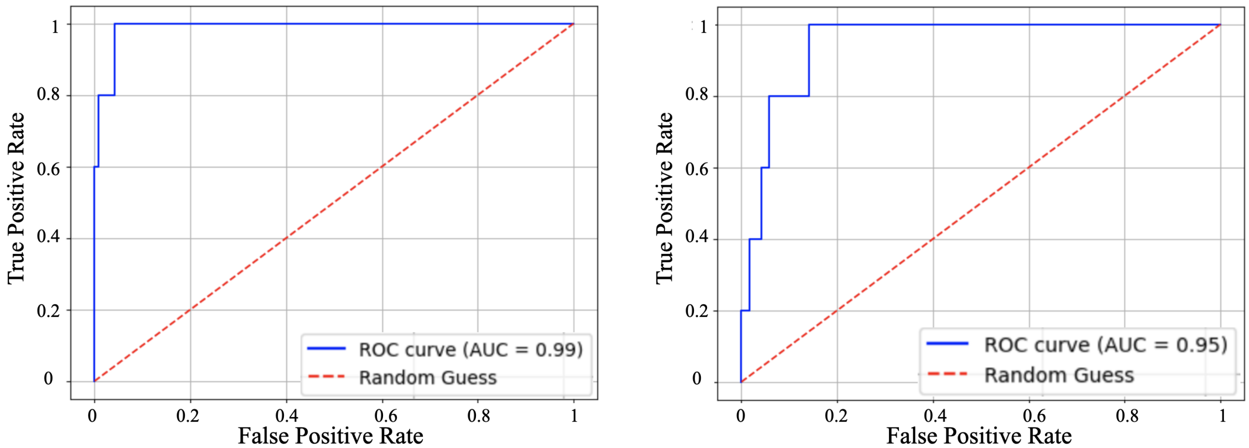


Figure 4.7: AUC-ROC curves demonstrating the classification model's performance in identifying top configurations for cost (left) and training time (right).

future deep learning model improvements.

Thus, we first present the results of off-the-shelf machine learning solutions applied to our dataset for accurate cost and time prediction as well as top candidate prediction. We then analyze feature importance using the Random Forest model, providing insights into the correlation between the most influential features and the labels in our dataset.

4.5.2.1 Traditional Machine Learning Model Training

To evaluate traditional machine learning algorithms for cost and time prediction, we experimented with Linear Regression [149], Random Forest Regressor [150], and XGBoost [88]. Our goal was to assess their effectiveness in capturing patterns and relationships within the data and compare their predictive performance against deep learning models.

We trained these models using an 80%-20% training-validation split and evaluated them with standard regression metrics: Root Mean Squared Error (RMSE) and R^2 score. The performance metrics for cost and training time prediction are summarized in Tables 4.4 and 4.5, respectively.

For the top candidate prediction task (identifying the best configurations based on cost and training time), we employed Random Forest for classification. Although we also experimented with

Table 4.4: Cost Prediction Machine Learning Models Performance Metrics (Validation Set)

Metric	**Linear Regression**	**Random Forest Regressor**	**XGBoost**
RMSE	1.57	1.36	1.49
R^2 **score**	0.95	0.96	0.96

Table 4.5: Training Time Prediction Machine Learning Models Performance Metrics (Validation Set)

Metric	**Poisson Regressor**	**Random Forest Regressor**	**XGBoost**
RMSE	1.78	1.12	0.5
R^2 **score**	0.88	0.95	0.99

Logistic Regression, Random Forest consistently outperformed it. Therefore, we focus on discussing the results obtained using Random Forest. It is important to note that the test set consists of an unseen training task (i.e., AG News [161]), comprising 126 data points that have no overlap with the training set.

Random Forest demonstrated strong performance in identifying optimal configurations in terms of cost. With an AUC of 0.94, the model successfully predicted 4 out of 5 top configurations while producing only one false positive, using a well-calibrated threshold to classify predictions as 1 (top candidate) or 0 (non-top candidate).

For training time-based top candidate prediction, Random Forest achieved an AUC of 0.88, which is sufficiently strong for this task. With an optimized threshold, the model correctly identified 4 out of 5 true positives, albeit with 17 false positives (over 83%). Despite the false positives, the model significantly reduced the number of potential configurations. This could be useful in combination with an approach like CherryPick that requires actual trial runs, enabling an efficient selection process based purely on input features.

Performance Observations: Linear Regression and Poisson Regressor exhibit significantly lower performance compared to both other machine learning models and our deep learning models. Their higher RMSE values and poor confusion plots indicate an inability to capture subtle variations and complex patterns in the data, making them less suitable for accurate predictions.

4.5.2.2 Feature Importance Analysis

To gain deeper insights into the dataset, we conducted a feature importance analysis using the Random Forest models, which demonstrated strong performance and acceptable generalization for both cost and training time prediction tasks. This analysis allowed us to identify the most influential features in our prediction models, as shown in Figure 4.8.

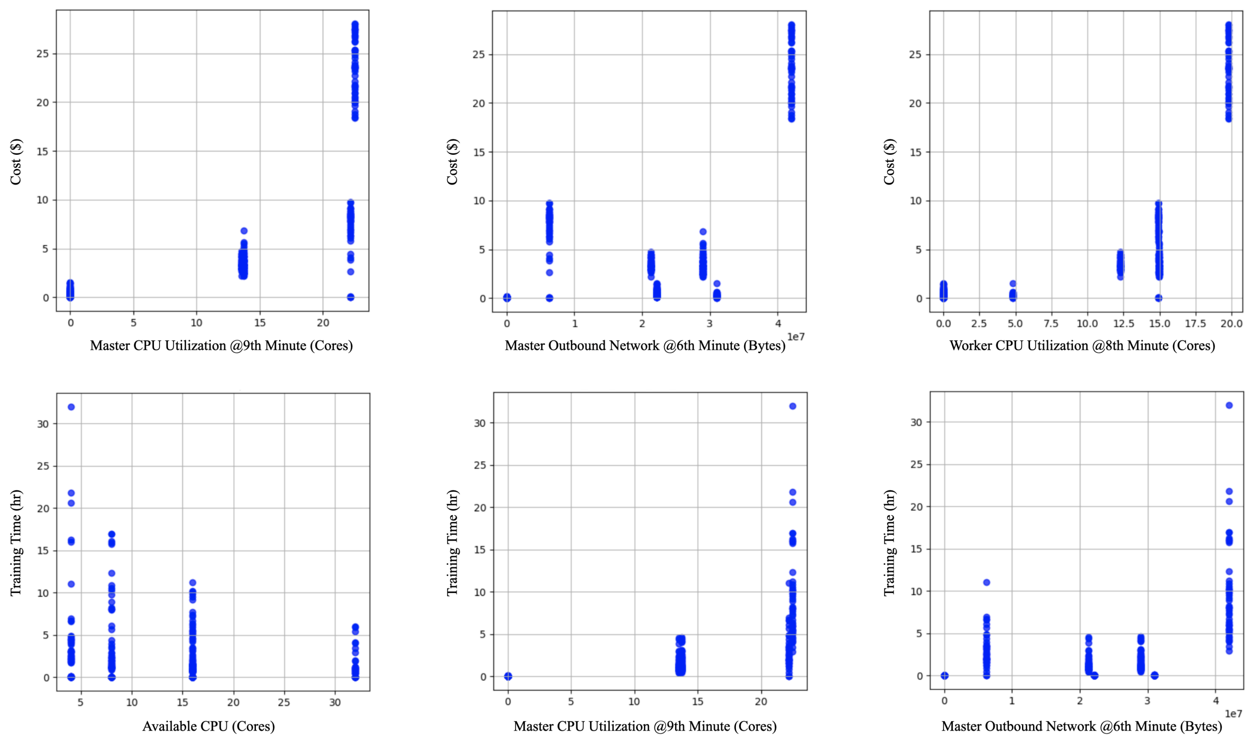


Figure 4.8: Distribution of top 3 features in accurate cost (top row) and training time (bottom row) prediction models using Random Forest Regressor Analyzer

Cost Prediction Insights: The cost prediction model highlights the significance of resource usage time series as key predictive features. Specifically, the top three features include CPU and network usage on both master and worker nodes at different time intervals during the training process on a local setup. As described in Section 4.3, the model incorporates up to 10 minutes of resource usage data to enhance prediction accuracy.

A closer examination of resource usage patterns—such as CPU utilization at the 9th minute on the master node, outbound network transfer at the 6th minute on the master node, and CPU utilization at the 8th minute on the worker nodes—reveals a distinct trend. Jobs with higher resource utilization (CPU and network in this case) in later intervals, rather than just at the start, tend to incur higher costs. This correlation may arise from heavier workloads or more computationally intensive models, which demand greater resource consumption and consequently lead to increased costs.

Training Time Prediction Insights: Similarly, the training time prediction model identifies several key factors influencing the duration of training. A significant observation is the inverse relationship between available CPU resources and training time: as CPU availability increases, training times tend to decrease. Furthermore, the correlation between CPU utilization at the 9th minute on the master node and outbound network transfer at the 6th minute with training time suggests that higher resource utilization during later stages of training is associated with longer training durations in this dataset.

Top Candidate Prediction: Our feature importance analysis using the Random Forest model for predicting top candidates reveals that batch size, number of workers, and number of machines are the three most influential features in classifying a configuration as top in terms of cost. Similarly, for training time, the most critical factors are the number of workers, number of machines, and available CPU resources. These findings highlight the significant role of resource allocation and system configuration in determining optimal performance.

Limitations: These findings offer valuable, data-driven insights tailored to the analyzed dataset. However, it is essential to recognize that these interpretations may not generalize across different datasets, computational environments, or job configurations. For example, the observed correlation between late-stage resource usage and increased costs or longer training times may vary in other settings.

Nevertheless, leveraging feature importance analysis enhances our understanding of key factors influencing cost and training time. These insights can guide resource allocation strategies and model optimization efforts, contributing to more efficient system configurations and improved performance.

4.6 Discussion & Future Directions

This research presents a solution for selecting optimal or near-optimal configurations for running distributed deep learning workloads on popular cloud platforms such as Google Vertex AI. While our approach has demonstrated promising results, several open challenges and future research directions remain.

Dataset Limitations Our dataset was constrained by funding limitations, allowing us to collect data from only seven applications with multiple configurations. These configurations were constructed by varying a subset of application hyperparameters and cloud infrastructure-related features. Expanding the dataset to include more applications, a broader range of configurations, and additional meaningful features could significantly enhance model performance. Furthermore, an interesting research direction would be exploring whether deep learning models can predict cost and training time for unseen models and training tasks, rather than focusing solely on identifying top-performing configurations.

Model Complexity The deep learning architecture used in this study was chosen for its effectiveness on our dataset. However, as the dataset expands—incorporating more features and diverse configuration combinations—new patterns and nonlinear relationships may emerge, requiring more sophisticated model architectures. Future work will explore deeper and more specialized neural network designs to better capture these complexities.

Potential directions include leveraging ensemble methods, model stacking, and multitask learning to improve predictive performance and generalization. Additionally, incorporating richer feature representations—such as detailed layer specifications (e.g., Conv2D dimensions) or embeddings generated from the training code using models like BERT [89]—could enhance the model's ability to make accurate and context-aware predictions.

Infrastructure Considerations This study focused on CPU-based training tasks and assumed the availability of resource usage records from training runs on local hardware. However, introducing GPUs or other specialized hardware into the training pipeline may introduce new challenges, such as managing GPU memory constraints, optimizing parallelism, and adapting to hardware-specific performance variations. Addressing these factors will be crucial for extending the applicability of this approach to a broader range of deep learning workloads.

These efforts will allow us to refine our approach and expand its applicability, ultimately contributing to more efficient cloud resource management in distributed deep learning workloads.

4.7 Related Work

There have been several efforts to determine the best cloud configuration for running various applications. CherryPick [67] attempts to find a near-optimal configuration by leveraging data from a few sample runs within the cloud, while Ernest [172] constructs a performance model from application characteristics to recommend an optimal setup. While effective, these methods are primarily designed for recurrent jobs, where collecting prior execution data is feasible. Additionally, solutions like CherryPick [67] face scalability challenges, as their computational cost increases significantly with the expansion of the configuration search space.

Other works have focused on autoscaling and resource reconfiguration to minimize waste and improve efficiency. OptimusCloud [123], PASCAL [119], and Escra [84] primarily address resource reallocation during execution. OptimusCloud [123] is tailored for database systems, incorporating database-specific features to optimize performance across heterogeneous infrastructures. PASCAL [119] introduces proactive autoscaling strategies that efficiently allocate resources to enhance response times while minimizing costs. However, despite their benefits, these solutions do not explicitly optimize for user preferences in distributed deep learning services like Google Vertex AI [99], where clients have greater flexibility in configuration choices. Additionally, DDL workloads are not highly dynamic, eliminating the need for continuous online monitoring and autoscaling for real-time resource optimization.

Lastly, prior research on cloud workload prediction has primarily focused on optimizing resource allocation through workload profiling, as seen in esDNN [178]. Additionally, DNNMem [96] and LLMem [111] have aimed to estimate the GPU memory required for efficiently running deep learning tasks. However, these solutions are either designed for cloud providers managing highly dynamic workloads or focus solely on memory allocation for specific tasks, overlooking other critical configuration parameters—such as application features—that significantly influence cost and training time optimization from a user perspective.

Our work complements these efforts by providing a predictive framework for selecting optimal configurations before executing a distributed deep learning workload in the cloud. This approach can help users make informed decisions, reducing the need for iterative tuning and improving overall efficiency.

4.8 Conclusion

In this research, we highlighted the increasing challenge of selecting optimal cloud configurations for distributed deep learning tasks as more organizations transition to AI services provided by major cloud platforms. Finding the right balance between cloud configurations and infrastructure is crucial to maximizing performance and minimizing costs. We proposed that by leveraging a minimal local run of a job along with application configuration and hardware information, we can design a deep learning solution to predict two critical metrics—training time and cost—when possible or rank best possible configurations and use these predictions to inform the selection of cloud configurations.

The results of our experiments on Google Vertex AI demonstrate the potential of our proposed solution, indicating that accurate predictions can help guide users toward more efficient cloud infrastructure choices.

Chapter 5

Conclusion

The rapid expansion of deep learning has brought both significant opportunities and substantial challenges to AI model development and deployment. As deep learning workloads grow in complexity and scale, optimizing resource allocation, ensuring system resilience, and managing cloud costs have become critical concerns. This thesis addresses these challenges by proposing novel solutions across three key areas:

- **Dynamic resource allocation**: We introduce *Escra*, a next-generation container orchestrator that performs sub-second, event-driven resource allocation. By continuously adapting resource distribution based on real-time utilization, Escra minimizes waste caused by overallocation and improves the efficiency of cloud-based machine learning workloads.

- **Fault-tolerant distributed training**: We develop a transparent, hardware-offloaded RDMA-based communication mechanism that enhances the reliability of distributed deep learning frameworks. This solution mitigates disruptions caused by network failures without requiring modifications to the underlying applications, ensuring robust performance in large-scale distributed training environments.

- **Cost-effective cloud configuration selection**: We propose a predictive cost and training time modeling approach that helps developers make informed decisions about cloud configurations. By analyzing lightweight, localized runs of deep learning tasks, our method provides accurate estimates of resource requirements, enabling users to balance performance and cost efficiency effectively.

Together, these contributions form a comprehensive framework for improving the efficiency, resilience, and cost-effectiveness of deep learning workloads in modern computing environments. The advancements presented in this thesis pave the way for more intelligent and adaptive deep learning infrastructure, reducing operational overhead and enabling scalable, high-performance AI systems.

Future Work

Several promising directions extend the research presented in this thesis:

- **Large-Scale Deployment:** Deploying the THORN system at scale is an exciting avenue for future work, as the growing demand for distributed deep learning infrastructure continuously introduces new challenges. Additionally, in the context of cost estimation and optimal configuration selection, training predictive models at the cloud provider level—where visibility into millions of training tasks is possible—could significantly enhance prediction accuracy. This broader scope would enable the development of more sophisticated model architectures and feature sets for workload analysis.

- **Infrastructure Adaptation:** While our configuration selection approach primarily focuses on CPU-based training, modern developers increasingly utilize GPU-equipped local machines for deep learning tasks. Extending our methodology to GPU-based infrastructures would provide deeper insights into performance variations and enable the discovery of similar predictive patterns across different hardware configurations.

- **Advanced Resource Management:** Systems like Escra are highly effective in reducing resource waste for dynamic workloads in datacenters. Since distributed deep learning (DDL) workloads exhibit predictable execution patterns, a natural extension of our work would involve analyzing deep learning code, extracting workload-specific features, and integrating them with infrastructure characteristics and local run resource utilization and by training predictive models on large-scale training task datasets, we could accurately estimate the resource utilization of DDL workloads in the cloud. This would provide cloud providers with valuable insights, improving decision-making in resource management and scheduling.

These future directions aim to refine and expand the impact of our contributions, paving the way for more efficient, scalable, and intelligent deep learning infrastructure.

Bibliography

[1] Accelerate. [Online; accessed 04-October-2024].

[2] Alibaba cluster trace program. https://github.com/alibaba/clusterdata.

[3] Amazon elastic container service. https://aws.amazon.com/ecs/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc.

[4] Apache openwhisk. https://github.com/apache/openwhisk.

[5] Assign memory resources to containers and pods. https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/.

[6] Aws lambda. https://aws.amazon.com/lambda/.

[7] Aws lambda enables functions that can run up to 15 minutes. https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/.

[8] Azure functions. https://azure.microsoft.com/en-us/services/functions/.

[9] Azure kubernetes service (aks). https://azure.microsoft.com/en-us/services/kubernetes-service/#overview.

[10] Bidirectional Forwarding Detection Commands on the Cisco IOS XR Software. [Online; accessed 04-October-2024].

[11] cadvisor. https://github.com/google/cadvisor.

[12] Cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[13] cifar10 — tensorflow datasets. [Online; accessed 04-October-2024].

[14] client-go. https://github.com/kubernetes/client-go.

[15] Cloud functions. https://cloud.google.com/functions.

[16] Cloud functions execution environment. https://cloud.google.com/functions/docs/concepts/exec#timeout.

[17] codeparrot/codeparrot - hugging face. [Online; accessed 04-October-2024].

[18] codeparrot/codeparrot-clear - datasets at hugging face. [Online; accessed 04-October-2024].

[19] Docker. https://www.docker.com/.

[20] Docker should assist bandwidth limiting containers. https://github.com/moby/moby/issues/26767.

[21] Ethernet switch device driver model (switchdev) – the linux kernel documentation. [Online; accessed 04-October-2024].

[22] FRRouting. [Online; accessed 04-October-2024].

[23] GitHub CoPilot – Your AI pair programmer. [Online; accessed 04-October-2024].

[24] GoBGP. [Online; accessed 04-October-2024].

[25] hey. https://github.com/rakyll/hey.

[26] Hipster shop: Cloud-native microservices demo application. https://github.com/Brown-NSG/microservices-demo.

[27] host.json reference for azure functions 2.x and later. https://docs.microsoft.com/en-us/azure/azure-functions/functions-host-json#functiontimeout.

[28] Hyperparameter tuning grid search example. https://github.com/lithops-cloud/applications/tree/master/sklearn.

[29] Ibm cloud functions. https://www.ibm.com/cloud/functions.

[30] Imdb. https://www.imdb.com/.

[31] Kubelet/kubernetes should work with swap enabled. https://github.com/kubernetes/kubernetes/issues/53533.

[32] Kubernetes: Production-grade container orchestration. https://kubernetes.io/.

[33] Kubernetes vertical pod autoscaler. https://github.com/prometheus/prometheus.

[34] Linux containers (lxc). https://linuxcontainers.org/.

[35] Lithops. https://lithops-cloud.github.io/.

[36] MSTFLINT Package - Firmware Burning and Diagnostics Tools. [Online; accessed 04-October-2024].

[37] netlink(7) — linux manual page. [Online; accessed 04-October-2024].

[38] Network function representors – the linux kernel documentation. [Online; accessed 04-October-2024].

[39] Network Lessons: Bidirectional Forwarding Detection (BFD). [Online; accessed 04-October-2024].

[40] Nvidia gpu direct. https://developer.nvidia.com/gpudirect.

[41] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[42] One year using kubernetes in production: Lessons learned. https://techbeacon.com/devops/one-year-using-kubernetes-production-lessons-learned.

[43] Openwhisk system details and limits. https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-limits.

[44] Performance co-pilot (pcp) manual. https://pcp.io/docs/index.html.

[45] Pytorch collective communication functions. https://pytorch.org/docs/stable/distributed.html#collectialbve-functions.

[46] Pytorch process group initialization. https://pytorch.org/docs/stable/distributed.html#initialization.

[47] Quotas and limit ranges. https://docs.openshift.com/online/pro/dev_guide/compute_resources.html.

[48] Ray Train. [Online; accessed 04-October-2024].

[49] Resource management guide - introduction to cgroups. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01.

[50] Resource quotas. https://kubernetes.io/docs/concepts/policy/resource-quotas/.

[51] rust-netlink. [Online; accessed 04-October-2024].

[52] Saving and loading a general checkpoint in pytorch. [Online; accessed 04-October-2024].

[53] Saving and loading checkpoints – ray 2.38.0. [Online; accessed 04-October-2024].

[54] scikit-learn: Machine learning in python. https://scikit-learn.org/stable/.

[55] Single Root IO Virtualization (SR-IOV) – NVIDIA Docs. [Online; accessed 04-October-2024].

[56] Teastore. https://github.com/DescartesResearch/TeaStore.

[57] Telecom at&t's paradise: 75% of telco's mpls tunnel data traffic now under sdn control. https://www.fiercetelecom.com/telecom/at-t-s-paradise-75-telco-s-mpls-tunnel-data-traffic-now-under-sdn-control.

[58] tf.keras.applications.resnet101 — tensorflow v2.16.1. [Online; accessed 04-October-2024].

[59] The BIRD Internet Routing Daemon. [Online; accessed 04-October-2024].

[60] Train ticket: A benchmark microservice system. https://github.com/FudanSELab/train-ticket.

[61] Training checkpoints — tensorflow core. [Online; accessed 04-October-2024].

[62] Vmware nsx data center. https://www.vmware.com/products/nsx.html.

[63] What is kubernetes. `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`.

[64] Marcelo Abranches, Sepideh Goodarzy, Maziyar Nazari, Shivakant Mishra, and Eric Keller. Shimmy: Shared memory channels for high performance inter-container communication. In 2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19), 2019.

[65] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 923–935, Boston, MA, July 2018. USENIX Association.

[66] Zaid Al-Ali, Sepideh Goodarzy, Ethan Hunter, Sangtae Ha, Richard Han, Eric Keller, and Eric Rozner. Making serverless computing more serverless. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pages 456–459, 2018.

[67] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 469–482, Boston, MA, March 2017. USENIX Association.

[68] Amazon. Amazon SageMaker. https://aws.amazon.com/pm/sagemaker/. [Online; accessed 24-October-2023].

[69] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In Proceedings of the third workshop on Hot topics in software defined networking, pages 1–6, 2014.

[70] Eric A Brewer. Kubernetes and the path to cloud native. In Proceedings of the Sixth ACM Symposium on Cloud Computing, pages 167–167, 2015.

[71] Broadcom. Broadcom Unveils Industry's Highest Performance Fabric for AI Networks. [Online; accessed 13-September-2024].

[72] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in AWS lambda. In 2023 USENIX Annual Technical Conference (USENIX ATC 23), pages 315–328, Boston, MA, July 2023. USENIX Association.

[73] Neil Brown. Control groups, part 4: On accounting. `https://lwn.net/Articles/606004/`.

[74] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. Queue, 14(1):70–93, 2016.

[75] Blake Caldwell, Sepideh Goodarzy, Sangtae Ha, Richard Han, Eric Keller, Eric Rozner, and Youngbin Im. Fluidmem: Full, flexible, and fast memory disaggregation for the cloud. In 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), pages 665–677, 2020.

[76] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. ACM SIGCOMM computer communication review, 37(4):1–12, 2007.

[77] Chia-Chen Chang, Shun-Ren Yang, En-Hau Yeh, Phone Lin, and Jeu-Yih Jeng. A kubernetes-based monitoring platform for dynamic cloud resource provisioning. In GLOBECOM 2017-2017 IEEE Global Communications Conference, pages 1–6. IEEE, 2017.

[78] Wei Chen, Aidi Pi, Shaoqi Wang, and Xiaobo Zhou. Pufferfish: Container-driven elastic memory management for data-intensive applications. In Proceedings of the ACM Symposium on Cloud Computing, pages 259–271, 2019.

[79] Cisco. Evolve your AI/ML Network with Cisco Silicon One. [Online; accessed 13-September-2024].

[80] Google Cloud. Vertex ai distributed training, 2025. Accessed: 2025-01-05.

[81] Google Cloud. Vertex ai pricing, 2025. Accessed: 2025-01-05.

[82] Cloudlab. Cloudlab Hardware Documentation. https://docs.cloudlab.us/hardware.html. [Online; accessed 24-October-2023].

[83] Nvidia Cooperation. Developing a linux kernel module using rdma for gpudirect, 2012.

[84] Greg Cusack, Maziyar Nazari, Sepideh Goodarzy, Erika Hunhoff, Prerit Oberai, Eric Keller, Eric Rozner, and Richard Han. Escra: Event-driven, sub-second container resource allocation. In 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS), pages 313–324, 2022.

[85] Greg Cusack, Maziyar Nazari, Sepideh Goodarzy, Prerit Oberai, Eric Rozner, Eric Keller, and Richard Han. Efficient microservices with elastic containers. In Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '19 Companion, page 65–67, New York, NY, USA, 2019. Association for Computing Machinery.

[86] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, NIPS'12, page 1223–1231, Red Hook, NY, USA, 2012. Curran Associates Inc.

[87] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. SIGPLAN Not., 49(4):127–144, February 2014.

[88] XGBoost Developers. Xgboost python api — xgboost 1.7.6 documentation, 2025. Accessed: 2025-01-05.

[89] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[90] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. arXiv preprint arXiv:2407.21783, 2024.

[91] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In Proceedings of the USENIX Annual Technical Conference (ATC), pages 1–14, July 2019.

[92] Dinesh G Dutt. EVPN in the Data Center. O'Reilly Media, Inc., 2018.

[93] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: an intellectual history of programmable networks. ACM SIGCOMM Computer Communication Review, 44(2):87–98, 2014.

[94] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 3–18, 2019.

[95] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. Rdma over ethernet for distributed training at meta scale. In Proceedings of the ACM SIGCOMM 2024 Conference, pages 57–70, 2024.

[96] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. Estimating gpu memory consumption of deep learning models. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, page 1342–1352, New York, NY, USA, 2020. Association for Computing Machinery.

[97] Sepideh Goodarzy, Maziyar Nazari, Richard Han, Eric Keller, and Eric Rozner. Resource management in cloud computing using machine learning: A survey. In 2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA), pages 811–816, 2020.

[98] Google. Google cloud functions.

[99] Google. Google Vertex AI Platform. https://cloud.google.com/vertex-ai. [Online; accessed 24-October-2023].

[100] Albert Greenberg, Gisli Hjalmtysson, David A Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4d approach to network control and management. ACM SIGCOMM Computer Communication Review, 35(5):41–54, 2005.

[101] Krzysztof Grygiel and Marcis Wielgus. Kubernetes vertical pod autoscaler. `https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/vertical-pod-autoscaler.md`.

[102] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In 2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS), pages 1–10. IEEE, 2019.

[103] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. ACM SIGCOMM Computer Communication Review, 40(4):195–206, 2010.

[104] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In Proceedings of the first workshop on Hot topics in software defined networks, pages 19–24, 2012.

[105] Patrick Helber, Benjamin Bischke, Andreas Dengel, and Damian Borth. Eurosat: A novel dataset and deep learning benchmark for land use and land cover classification, 2019.

[106] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In NSDI, volume 11, pages 22–22, 2011.

[107] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[108] HuggingFace. HuggingFace Model Parallelism. [Online; accessed 18-July-2024].

[109] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level {TCP} stack for multicore systems. In 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14), pages 489–502, 2014.

[110] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating network-based CPU in container environments. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 313–328, Renton, WA, April 2018. USENIX Association.

[111] Taeho Kim, Yanming Wang, Vatshank Chaturvedi, Lokesh Gupta, Seyeon Kim, Yongin Kwon, and Sangtae Ha. Llmem: Estimating gpu memory usage for fine-tuning pre-trained llms, 2024.

[112] Alexey Kopytov. Sysbench. `https://github.com/akopytov/sysbench`.

[113] Kubeflow. Kubeflow training operator, 2025. Accessed: 2025-01-05.

[114] Kubernetes IN Docker (KIND). Kubernetes in docker (kind), 2025. Accessed: 2025-01-05.

[115] Leslie Lamport and K. Mani Chandy. Distributed snapshots: Determining global states of a distributed system. ACM Transactions on Computer Systems, 3(1), Feb. 1985.

[116] Kevin Lee, Adi Gangidi, and Mathew Oldham. Building meta's genai infrastructure. `https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/`, Mar. 2024.

[117] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 583–598, Broomfield, CO, October 2014. USENIX Association.

[118] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training, 2020.

[119] Federico Lombardi, Andrea Muti, Leonardo Aniello, Roberto Baldoni, Silvia Bonomi, and Leonardo Querzoni. Pascal: An architecture for proactive auto-scaling of distributed services. Future Gener. Comput. Syst., 98(C):342–361, September 2019.

[120] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In 2017 IEEE International Conference on Big Data (Big Data), pages 2884–2892. IEEE, 2017.

[121] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-Path transport for RDMA in datacenters. In USENIX Symposium on Networked Systems Design and Implementation (NSDI), April 2018.

[122] Mallik Mahalingam, Dinesh G. Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T. Sridhar, Mike Bursell, and Chris Wright. RFC 7348: Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks, Aug. 2014. [Online; accessed 04-October-2024].

[123] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 189–203. USENIX Association, July 2020.

[124] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev., 38(2):69–74, March 2008.

[125] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[126] Maziyar Nazari, Sepideh Goodarzy, Eric Keller, Eric Rozner, and Shivakant Mishra. Optimizing and extending serverless platforms: A survey. In 2021 Eighth International Conference on Software Defined Systems (SDS), pages 1–8, 2021.

[127] NVIDIA. NCCL. [Online; accessed 18-September-2024].

[128] NVIDIA. NCCL Tests. [Online; accessed 18-September-2024].

[129] NVIDIA. NVIDIA Spectrum-X Network Platform Architecture. [Online; accessed 13-September-2024].

[130] Openwhisk. Apache Openwhisk: Open Source Serverless Cloud Platform. https://openwhisk.apache.org/. [Online; accessed 24-October-2023].

[131] Amy Ousterhout, Adam Belay, and Irene Zhang. Just in time delivery: Leveraging operating systems knowledge for better datacenter congestion control. In 11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19), 2019.

[132] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In 16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19), pages 361–378, 2019.

[133] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In Proceedings of ACM SIGCOMM 2022, 2022.

[134] Prometheus. Prometheus: Open-source monitoring and alerting toolkit, 2025. Accessed: 2025-01-05.

[135] Pytorch. Pytorch Distributed Data Parallelism. [Online; accessed 18-July-2024].

[136] Pytorch. Pytorch Pipeline Parallelism API. https://pytorch.org/docs/stable/pipeline.html. [Online; accessed 24-October-2023].

[137] Pytorch. Pytorch Python Package. https://pypi.org/project/torch/. [Online; accessed 24-October-2023].

[138] Pytorch. Torchvision mnist dataset.

[139] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 805–825. USENIX Association, November 2020.

[140] Robi Rahman, David Owen, and Josh You. Tracking large-scale ai models. `https://epochai.org/blog/tracking-large-scale-ai-models`, Apr. 2024.

[141] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020.

[142] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pages 33–40. IEEE, 2019.

[143] Renato J. Recio, Bernard Metzler, Paul R. Culley, Jeff Hilland, and Dave Garcia. A remote direct memory access protocol specification. Technical Report RFC 5040, Internet Engineering Task Force, October 2007.

[144] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In Proceedings of the Third ACM Symposium on Cloud Computing, pages 1–13, 2012.

[145] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In Proceedings of the third ACM symposium on cloud computing, pages 1–13, 2012.

[146] Benoit Rostykus and Gabriel Hartmann. Predictive cpu isolation of containers at netflix. https://netflixtechblog.com/predictive-cpu-isolation-of-containers-at-netflix-91f014d856c7.

[147] Krzysztof Rzadca, Paweł Findeisen, Jacek Świderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Paweł Krzysztof Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at google scale. In Proceedings of the Fifteenth European Conference on Computer Systems, 2020.

[148] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In Proceedings of the 8th ACM European Conference on Computer Systems, pages 351–364, 2013.

[149] Scikit-learn. Linearregression — scikit-learn 1.5 documentation, 2025. Accessed: 2025-01-05.

[150] Scikit-learn. Randomforestregressor — scikit-learn 1.5 documentation, 2025. Accessed: 2025-01-05.

[151] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

[152] Hemal Shah, Felix Marti, Wael Noureddine, Asgeir Eiriksson, and Robert Sharp. Remote direct memory access RDMA protocol extensions. Technical Report RFC 7306. ISSN: 2070-1721, Internet Engineering Task Force, February 2014.

[153] Jay Shah and Dushyant Dubaria. Building modern clouds: using docker, kubernetes & google cloud platform. In 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), pages 0184–0189. IEEE, 2019.

[154] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 205–218. USENIX Association, July 2020.

[155] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. Resource deflation: A new approach for transient resource reclamation. In Proceedings of the Fourteenth EuroSys Conference 2019, pages 1–17, 2019.

[156] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-tensorflow: Deep learning for supercomputers, 2018.

[157] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. CoRR, abs/1909.08053, 2019.

[158] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[159] John Sonchack, Jonathan M Smith, Adam J Aviv, and Eric Keller. Enabling practical software-defined networking security applications with ofx. In NDSS, volume 16, pages 1–15, 2016.

[160] Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, and Mun Choon Chan. Network load balancing with in-network reordering support for rdma. In Proceedings of the ACM SIGCOMM 2023 Conference, pages 816–831, 2023.

[161] PyTorch Team. Ag news dataset module, 2025. Accessed: 2025-02-20.

[162] PyTorch Team. Eurosat dataset module, 2025. Accessed: 2025-02-20.

[163] PyTorch Team. Imdb dataset module, 2025. Accessed: 2025-02-20.

[164] PyTorch Team. Udpos dataset module, 2025. Accessed: 2025-02-20.

[165] PyTorch Team. Yesno dataset module, 2025. Accessed: 2025-02-20.

[166] Gil Tene. wrk2: a http benchmarking tool based mostly on wrk. `https://github.com/giltene/wrk2`.

[167] Tensorflow. Tensorflow Distributed Data Parallel API. https://www.tensorflow.org/guide/distributed_training. [Online; accessed 24-October-2023].

[168] Tensorflow. Tensorflow Multiworker Mirrored Strategy. [Online; accessed 18-July-2024].

[169] Feng Tian, Yang Zhang, Wei Ye, Cheng Jin, Ziyan Wu, and Zhi-Li Zhang. Accelerating distributed deep learning using multi-path rdma in data center networks. In Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR), SOSR '21, page 88–100, New York, NY, USA, 2021. Association for Computing Machinery.

[170] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[171] Paul Turner, Bharata B Rao, and Nikhil Rao. Cpu bandwidth control for cfs. In Proceedings of the Linux Symposium, pages 245–254, 2010.

[172] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for Large-Scale advanced analytics. In 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), pages 363–378, Santa Clara, CA, March 2016. USENIX Association.

[173] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In Proceedings of the European Conference on Computer Systems (EuroSys), Bordeaux, France, 2015.

[174] Pablo Villalobos and Anson Ho. Trends in training dataset sizes. https://epochai.org/blog/trends-in-training-dataset-sizes, Sept. 2022.

[175] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In 2015 10th Computing Colombian Conference (10CCC), pages 583–590. IEEE, 2015.

[176] Yi Wang, Ya-nan Jiang, Qiufang Ma, Chen Tian, Bo Bai, and Gong Zhang. Rdma load balancing via data partition. In 2019 28th International Conference on Computer Communication and Networks (ICCCN), pages 1–8. IEEE, 2019.

[177] Ertza Warraich, Omer Shabtai, Khalid Manaa, Shay Vargaftik, Yonatan Piasetzky, Matty Kadosh, Lalith Suresh, and Muhammad Shahbaz. Ultima: Robust and tail-optimal allreduce for distributed deep learning in the cloud, 2023.

[178] Minxian Xu, Chenghao Song, Huaming Wu, Sukhpal Singh Gill, Kejiang Ye, and Chengzhong Xu. esdnn: Deep neural network based multivariate workload prediction in cloud computing environments. ACM Trans. Internet Technol., 22(3), August 2022.

[179] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast distributed deep learning over rdma. In Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[180] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In Proceedings of the ACM Symposium on Cloud Computing, SoCC '20. Association for Computing Machinery, 2020.

[181] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. ACM SIGCOMM Computer Communication Review, 45(4):523–536, 2015.

[182] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems. In Proceedings of the 2021 ACM SIGCOMM 2021 Conference, SIGCOMM '21, page 641–656, New York, NY, USA, 2021. Association for Computing Machinery.

[183] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, et al. Resiliency at scale: Managing {Google's}{TPUv4} machine learning supercomputer. In USENIX Symposium on Networked Systems Design and Implementation (NSDI), pages 761–774, 2024.