Efficient Resource Management for Machine Learning

by

Romil Bhardwaj

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Dr. Ganesh Ananthanarayanan
Professor Joseph E. Gonzalez
Professor Scott Shenker

Fall 2023

The dissertation of Romil Bhardwaj, titled Efficient Resource Management for Machine Learning, is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

_____ Date _____

University of California, Berkeley

Efficient Resource Management for Machine Learning

Abstract

Efficient Resource Management for Machine Learning

by

Romil Bhardwaj

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Invasive brag; forbearance.

To my family

Contents

List of Figures

List of Tables

Acknowledgments

Thanks!

Chapter 1

Introduction

Machine Learning (ML) is a major driver of economic value in today. Estimates suggest that ML technologies could potentially contribute up to 3 trillion USD [**something**] to the global economy in the coming years, impacting sectors ranging from healthcare to finance. Romil: Fluffy? Add more here.

Machine learning workloads, characterized by their intensive resource requirements, require multiple iterations over large datasets to generate progressive accurate models. This process demands significant computational power. As illustrated in Figure 1.1, the compute demand of ML models has escalated over time. For instance, training Llama 2 in 2023 required FILL ME petaFLOPs, representing an increase of FILL ME times over AlexNet, the largest model in 2012. This trend is indicative of the rising computational needs for ML, placing increasing pressure on existing resources.

On the other hand, the supply of compute is struggling to keep up with the demand. Moore's law, which postulates that the number of transistors on a chip doubles every two years, is hitting

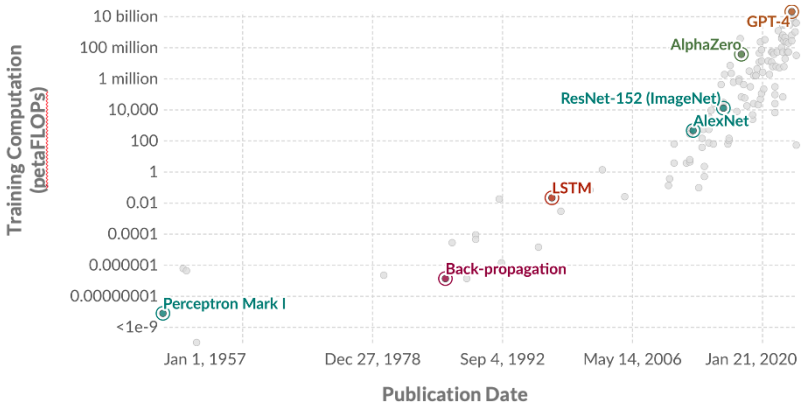


Figure 1.1: The compute demand of ML models over time. FILL ME

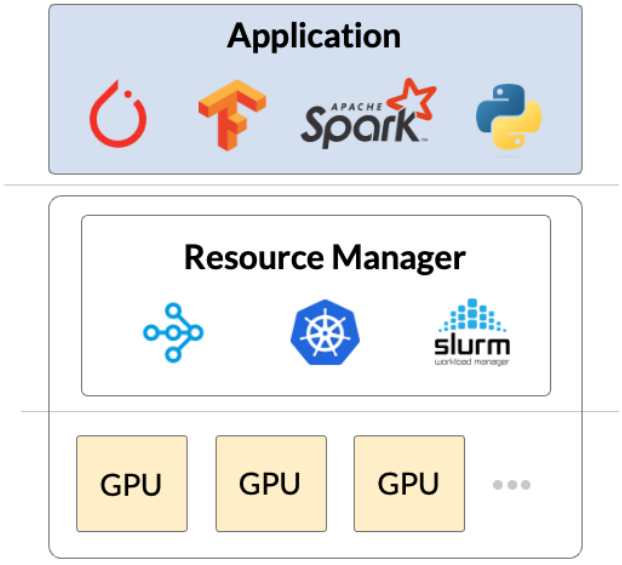


Figure 1.2: The machine learning stack spanning across application, cluster manager, and accelerator layers. FILL ME

a wall. As transistors become smaller, reaching the size of a few nanometers, physical limitations such as quantum effects and heat dissipation become significant challenges. This makes it increasingly difficult to continue shrinking components at the pace predicted by Moore's Law. Worse yet, Dennard scaling, which refers to the shrinking of transistors while maintaining constant power density, has ended because of increased leakage currents and heat dissipation issues at smaller transistor sizes. Combined, these factors have led to a plateau in the growth of compute performance.

The growing compute needs of Machine Learning and the shrinking supply of compute for these models has created a supply-demand gap. This gap manifests in two forms - lack of resource availability and high costs of training ML models. Romil: AddMore - Availability - use the news article links, Cost - quote from epochai study

In this thesis, we bridge the compute supply-demand gap by developing techniques to make machine learning more resource efficient. By optimizing resource utilization, we aim to reduce the overall demand for computational power, thereby easing the pressure on the strained resource supply.

How do we improve resource efficiency? Examining the ML stack reveals opportunities across its three layers (Figure 1.2):

1. **Application Layer:** This layer comprises ML models and their training algorithms. Tools like PyTorch and TensorFlow are commonly used for model definition and workflow specification.

2. **Cluster Manager:** Responsible for managing machine clusters for ML training, this layer oversees job scheduling, resource allocation, and progress tracking, with systems like Kubernetes and Mesos.

3. **Specialized Accelerators:** This hardware layer, including GPUs and TPUs, accelerates ML training, managed by lower-level schedulers.

In the accelerator layer, we introduce *Ekya* (Chapter FILL ME), enhancing the efficiency of continuous learning for high-accuracy inference by 4x. Enabled by the Thief Scheduling algorithm, Ekya innovatively reallocates resources among jobs, focusing on configurations that maximize accuracy improvements for resource costs. The Microprofiler complements this by accurately assessing resource demands and potential accuracy gains, allowing for informed and efficient scheduling decisions. This approach effectively manages resource constraints and addresses data drift challenges.

At the cluster manager layer, **Cilantro** (Chapter FILL ME) revolutionizes resource allocation by replacing inefficient human-estimated requests with an online learning mechanism. This model forms feedback loops with jobs, creating dynamic resource-to-performance mappings. Cilantro's uncertainty-aware policies, demonstrated in diverse settings, mark a paradigm shift towards more flexible, accurate, and efficient resource allocation.

Lastly, at the application layer, **ESCHER** (Chapter FILL ME) addresses the need for scheduling flexibility without complicating the underlying cluster manager. It introduces ephemeral resources, allowing applications to specify custom scheduling constraints easily. Implemented in Kubernetes and Ray, ESCHER demonstrates enhanced policy expression capabilities, balancing ease-of-use with customizability.

In conclusion, this thesis presents a comprehensive approach to addressing the compute supply-demand gap in ML. By innovating across the ML stack, it contributes significantly to more efficient resource utilization, paving the way for sustainable and cost-effective ML practices in the face of ever-growing computational demands.

Romil: Consider summarizing key contributions and previewing chapters

Chapter 2

Ekya: Efficient Continuous Learning on the Edge

Video analytics applications use edge compute servers for processing videos. Compressed models that are deployed on the edge servers for inference suffer from *data drift* where the live video data diverges from the training data. Continuous learning handles data drift by periodically retraining the models on new data. Our work addresses the challenge of jointly supporting inference and retraining tasks on edge servers, which requires navigating the fundamental tradeoff between the retrained model's accuracy and the inference accuracy. Our solution `Ekya` balances this tradeoff across multiple models and uses a micro-profiler to identify the models most in need of retraining. `Ekya`'s accuracy gain compared to a baseline scheduler is 29% higher, and the baseline requires $4\times$ more GPU resources to achieve the same accuracy as `Ekya`.

2.1 Introduction

Video analytics applications, such as for urban mobility [146, 54] and smart cars [49], are being powered by deep neural network (DNN) models for object detection and classification, e.g., Yolo [73], ResNet [77] and EfficientNet [114]. Video analytics deployments stream the videos to *edge servers* [15, 13] placed on-premise [7, 75, 160, 156]. Edge computation is preferred for video analytics as it does not require expensive network links to stream videos to the cloud [156], while also ensuring privacy of the videos (e.g., many European cities mandate against streaming their videos to the cloud [168, 4]).

Edge compute is provisioned with limited resources (e.g., with weak GPUs [13, 15]). This limitation is worsened by the mismatch between the growth rate of the compute demands of models and the compute cycles of processors [173, 5]. As a result, edge deployments rely on *model compression* [166, 187, 132]. The compressed DNNs are initially trained on representative data from each video stream, but while in the field, they are affected by *data drift*, i.e., the live video data diverges significantly from the data that was used for training [101, 149, 151, 33]. Cameras in streets and smart cars encounter varying scenes over time, e.g., lighting, crowd densities, and

changing object mixes. It is difficult to exhaustively cover all these variations in the training, especially since even subtle variations affect the accuracy. As a result, there is a sizable drop in the accuracy of edge DNNs due to data drift (by 22%; §2.2). In fact, the fewer weights and shallower architectures of compressed DNNs often make them unsuited to provide high accuracy when trained with large variations in the data.

Continuous model retraining. A promising approach to address data drift is continuous learning. The edge DNNs are incrementally *retrained* on new video samples even as some earlier knowledge is retained [158, 52]. Continuous learning techniques retrain the DNNs periodically [186, 141]; we refer to the period between two retrainings as the "retraining window" and use a sample of the data that is accumulated during each window for retraining. Such ongoing learning [192, 170, 86] helps the compressed models maintain high accuracy.

Edge servers use their GPUs [15] for DNN inference on many live video streams (e.g., traffic cameras in a city). Adding continuous training to edge servers presents a tradeoff between the live inference accuracy and drop in accuracy due to data drift. Allocating more resources to the retraining job allows it to finish faster and provide a more accurate model sooner. At the same time, during the retraining, taking away resources from the inference job lowers its accuracy (because it may have to sample the frames of the video to be analyzed).

Central to the resource demand and accuracy of the jobs are their *configurations*. For retraining jobs, configurations refer to the hyperparameters, e.g., number of training epochs, that substantially impact the resource demand and accuracies (§2.3). The improvement in accuracy due to retraining also depends on *how much* the characteristics of the live videos have changed. For inference jobs, configurations like frame sampling and resolution impact the accuracy and resources needed to keep up with analyzing the live video [74, 34].

Problem statement. We make the following decisions for retraining. (1) in each retraining window, decide which of the edge models to retrain; (2) allocate the edge server's GPU resources among the retraining and inference jobs, and (3) select the configurations of the retraining and inference jobs. We also constraint our decisions such that the inference accuracy *at any point in time* does not drop below a minimum value (so that the outputs continue to remain useful to the application). Our objective in making the above three decisions is to maximize the inference accuracy *averaged over the retraining window* (aggregating the accuracies during and after the retrainings). Maximizing inference accuracy over the retraining window creates new challenges as it is different from (i) video inference systems that optimize only the instantaneous accuracy [63, 34, 74], (ii) model training systems that optimize only the eventual accuracy [1, 18, 164, 169, 188, 135].

Addressing the fundamental tradeoff between the retrained model's accuracy and the inference accuracy is computationally complex. First, the decision space is multi-dimensional consisting of a diverse set of retraining and inference configurations, and choices of resource allocations over time. Second, it is difficult to know the performance of different configurations (in resource usage and accuracy) as it requires actually retraining using different configurations. Data drift exacerbates these challenges because a decision that works well in a retraining window may not do so in the future.

Solution components. Our solution `Ekya` has two main components: a resource scheduler and a performance estimator.

In each retraining window, the **resource scheduler** makes the three decisions listed above in our problem statement. In its decisions, Ekya's scheduler prioritizes retraining the models of those video streams whose characteristics have changed the most because these models have been most affected by data drift. The scheduler decides against retraining the models which do not improve our target metric. To prune the large decision space, the scheduler uses the following techniques. First, it simplifies the spatial complexity by considering GPU allocations only in coarse fractions (e.g., 10%) that are accurate enough for the scheduling decisions, while also being mindful of the granularity achievable in modern GPUs [127]. Second, it does not change allocations to jobs *during the retraining*, thus largely sidestepping the temporal complexity. Finally, our micro-profiler (described below) prunes the list of configurations to only the promising options.

To make efficient choices of configurations, the resource scheduler relies on estimates of accuracy after the retraining and the resource demands. We have designed a **micro-profiler** that observes the accuracy of the retraining configurations on a *small subset* of the training data in the retraining window with *just a few epochs*. It uses these observations to extrapolate the accuracies when retrained on a larger dataset for many more epochs. Further, we restrict the micro-profiling to only a small set of *promising* retraining configurations. These techniques result in Ekya's micro-profiler being $100\times$ more efficient than exhaustive profiling while still estimating accuracies with an error of 5.8%. To estimate the resource demands, the micro-profiler measures the retraining duration *per epoch* when 100% of the GPU is allocated, and scales for different allocations, epochs, and training data sizes.

Implementation and Evaluation. We have evaluated Ekya using a system implementation and trace-driven simulation. We used video workloads from dashboard cameras of smart cars (Waymo [133] and Cityscapes [110]) as well as from traffic and building cameras over 24 hours. Ekya's accuracy compared to competing baselines is 29% higher. As a measure of Ekya's efficiency, attaining the same accuracy as Ekya will require $4\times$ more GPU resources on the edge for the baseline.

Contributions: Our work makes the following contributions.

1) We introduce the metric of *inference accuracy averaged over the retraining window* for continuous training systems.

2) We design an *efficient micro-profiler to estimate* the benefits and costs of retraining edge DNN models.

3) We design a scalable resource scheduler for *joint retraining and inference* on edge servers.

2.2 Continuous training on edge compute

Edge Computing for Video Analytics

Video analytics deployments commonly analyze videos on edge servers placed on-premise (e.g., from AWS [13] or Azure [15]). A typical edge server supports tens of video streams [26], e.g., on the cameras in a building, with customized models for each stream [112] (see Figure 2.1).Video analytics applications adopt edge computing for the following reasons [156, 75, 7].

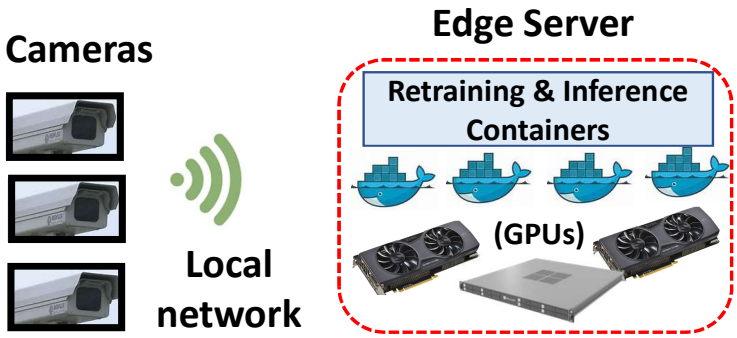


Figure 2.1: Cameras connect to the edge server, with consumer-grade GPUs for DNN inference and retraining containers.

1) Edge deployments are often in locations where the *uplink network to the cloud is expensive* for shipping continuous video streams, e.g., in oil rigs with expensive satellite network or smart cars with data-limited cellular network. *

2) Network links out of the edge locations experience *outages* [148, 156]. Edge compute provides robustness against disconnection to the cloud [42] and prevents disruptions [28].

3) Videos often contain *sensitive and private data* that users do not want sent to the cloud (e.g., many EU cities legally mandate that traffic videos be processed on-premise [168, 4]).

Thus, due to reasons of network cost and video privacy, it is preferred to run both inference and retraining on the edge compute device itself without relying on the cloud. In fact, with bandwidths typical in edge deployments, cloud-based solutions are slower and result in lower accuracies (§2.6).

Compressed DNN Models and Data drift

Advances in computer vision research have led to high-accuracy DNN models that achieve high accuracy with a large number of weights, deep architectures, and copious training data. While highly accurate, using these heavy and general DNNs for video analytics is both expensive and slow [34, 67], which make them unfit for resource-constrained edge computing. The most common approach to addressing the resource constraints on the edge is to train and deploy *specialized and compressed* DNNs [166, 187, 132, 102, 126, 113], which consist of far fewer weights and shallower architectures. For instance, Microsoft's edge video analytics platform [146] uses a compressed DNN (TinyYOLO [145]) for efficiency. Similarly, Google released Learn2Compress[54] for edge devices to automate the generation of compressed models from proprietary models. These compressed DNNs are trained to only recognize the limited objects and scenes specific to each video stream. In other words, to maintain high accuracy, they forego generality for improved compute efficiency [34, 67, 141].

*The uplinks of LTE cellular or satellite links is $3-10$Mb/s [111, 128], which can only support a couple of 1080p 30 fps HD video streams whereas a typical deployment has many more cameras [156].

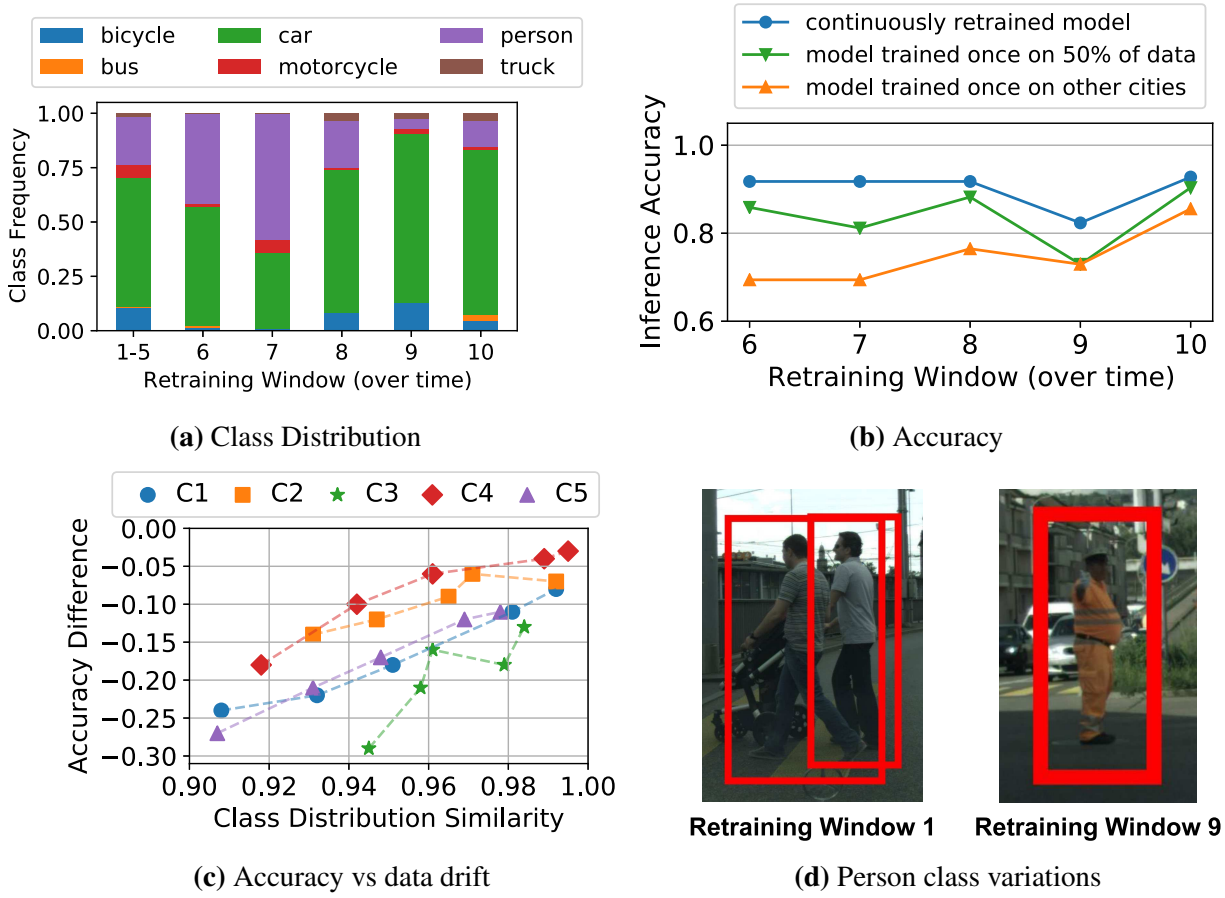
Data drift. As specialized edge DNNs have shallower architectures than general DNNs, they can only memorize limited amount of object appearances, object classes, and scenes. As a result, specialized edge DNNs are particularly vulnerable to *data drift* [101, 149, 151, 33], where live video data diverges significantly from the initial training data. For example, variations in the object pose, scene density (e.g. rush hours), and lighting (e.g., sunny vs. rainy days) over time make it difficult for traffic cameras to accurately identify the objects of interest (cars, bicycles, road signs). Cameras in modern cars observe vastly varying scenes (e.g., building types, crowd sizes) as they move through different neighborhoods and cities. Further, the *distribution* of the objects change over time, which reduces the edge model's accuracy [186, 202]. Due to their ability to memorize limited amount of object variations, edge DNNs have to be continuously updated with recent data and changing object distributions to maintain a high accuracy.

Continuous training. The preferred approach, that has gained significant attention, is for edge DNNs to *continuously learn* as they incrementally observe new samples over time [192, 170, 86]. The high temporal locality of videos allows the edge DNNs to focus their learning on the most recent object appearances and object classes [157, 141]. In `Ekya`, we use a modified version of iCaRL[170] learning algorithm to on-board new classes, as well as adapt to the changing characteristics of the existing classes. Since manual labeling is not feasible for continuous training systems on the edge, the labels for the retraining are obtained from a "golden model" - a highly accurate (87% and 84% accuracy on Cityscapes and Waymo datasets, respectively) but expensive model (deeper architecture with large number of weights). The golden model cannot keep up with inference on the live videos and we use it to label only a small fraction of the videos in the retraining window. Our approach is essentially that of supervising a low-cost "student" model with a high-cost "teacher" model (or knowledge distillation [66]), and this has been broadly applied in computer vision literature [192, 141, 86, 186].

Accuracy benefits of continuous learning

To show the benefits of continuous learning, we use the video stream from one example city in the Cityscapes dataset [110] that consists of videos from dashboard cameras in many cities. In our evaluation in §2.6, we use both moving dashboard cameras as well as static cameras over long time periods. We divide the video data in our example city into ten fixed *retraining windows* (200s in this example).

Understanding sources of data drift. Figure 2.2a shows the change of object class distributions across windows. The initial five windows see a fair amount of persons and bicycles, but bicycles rarely show up in windows 6 and 7, while the share of persons varies considerably across windows $6-10$. Figure 2.2c summarizes the effect of this data drift on model accuracy in five independent video streams, C1-C5. For each stream, we train a baseline model on the first five windows, and test it against five windows in the future and use cosine similarity to measure the class distribution shift for each window. Though accuracy generally improves when the model is used on windows with similar class distributions (high cosine similarity), the relationship is not guaranteed (C2, C3). This is because class distribution shift is not the only form of data drift. Illumination, pose and



(a) Class Distribution

(b) Accuracy

(c) Accuracy vs data drift

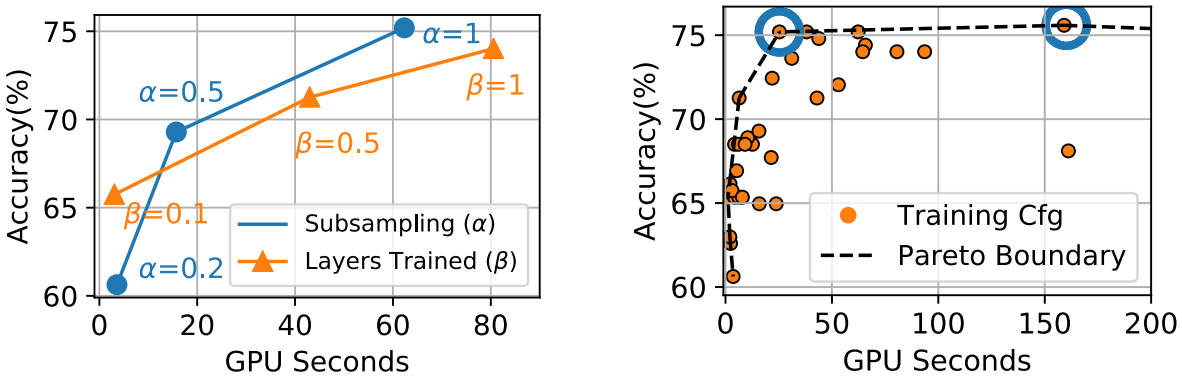
(d) Person class variations

Figure 2.2: Continuous learning in the Cityscapes dataset. Shift in class distributions (a) across windows necessitates continuous learning (b). Model accuracy is not only affected by class distribution shifts (c), but also by changes in object appearances (d).

appearance differences also affect model performance (e.g. clothing and angles for objects in the person class vary significantly; Figure 2.2d).

Improving accuracy with continuous learning. Figure 2.2b plots inference accuracy of an edge DNN (a compressed ResNet18 classifier) in the last five windows using different training options. (1) Training a compressed ResNet18 with video data on all other cities of the Cityscapes dataset does not result in good performance. (2) Unsurprisingly, we observe that training the edge DNN once using data from the first five windows *of this example city* improves the accuracy. (3) *Continuous retraining* using the most recent data for training achieves the highest accuracy consistently. Its accuracy is higher than the other options by up to 22%.

Interestingly, using the data from the first five windows to train the larger ResNet101 DNN (not graphed) achieves better accuracy than the continuously retrained ResNet18. The substantially better accuracy of ResNet101 compared to ResNet18 when trained *on the same data* of the first five



(a) Effect of Hyperparameters

(b) Resource-accuracy

Figure 2.3: Measuring retraining configurations. GPU seconds refers to the duration taken for retraining with 100% GPU allocation. (a) varies two example hyperparameters, keeping others constant. Note the Pareto boundary of configurations in (b); for every non-Pareto configuration, there is at least one Pareto configuration that is better than it in *both* accuracy and GPU cost.

windows also shows that this training data was indeed fairly representative. But the lightweight ResNet18's weights and architecture limits its ability to learn and is a key contributor to its lower accuracy. Nonetheless, ResNet101 is $13\times$ slower than the compressed ResNet18 [29]. This makes the efficient ResNet18 more suited for edge deployments and continuous learning enables it to maintain high accuracy even with data drift. Therefore, the need for continuous training of edge DNNs is ongoing and not just during a "ramp-up" phase.

2.3 Scheduling retraining and inference jointly

We propose *joint retraining and inference* on edge servers. The joint approach utilizes resources better than statically provisioning compute for retraining. Since retraining is periodic [186, 141] with far higher compute demands than inference, static provisioning causes idling. Compared to uploading videos to the cloud for retraining, our approach has advantages in privacy (§2.2), and network costs and accuracy (§2.6).

Configuration diversity of retraining and inference

Tradeoffs in retraining configurations. The hyperparameters for retraining, or "retraining configurations", influence the resource demands and accuracy. Retraining fewer layers of the DNN (or, "freezing" more layers) consumes lesser GPU resources, as does training on fewer data samples, but they also produce a model with lower accuracy; Figure 2.3a.

Configuration	Retraining Window 1		Retraining Window 2	
	End Accuracy	GPU seconds	End Accuracy	GPU seconds
Video A Cfg1A	75	85	95	90
Video A Cfg2A (*)	70	65	90	40
Video B Cfg1B	90	80	98	80
Video B Cfg2B (*)	85	50	90	70

Table 2.1: Hyperparameter configurations for retraining jobs in Figure 2.4's example. At the start of retraining window 1, camera A's inference model has an accuracy of 65% and camera B's inference model has an accuracy of 50%. Asterisk (*) denotes the configurations picked in Figures 2.4b and 2.4d.

Figure 2.3b illustrates the resource-accuracy trade-offs for an edge DNN (ResNet18) with various hyperparameters: number of training epochs, batch sizes, number of neurons in the last layer, number of frozen layers, and fraction of training data. We make two observations. First, there is a wide spread in the resource usage (measured in GPU seconds), by upto a factor of $200\times$. Second, higher resource usage does not always yield higher accuracy. For the two configurations circled in Figure 2.3b, their GPU demands vary by $6\times$ even though their accuracies are the same ($\sim 76\%$). Thus, careful selection of the configurations considerably impacts the resource efficiency. Moreover, the accuracy spread across configurations is dependent on the extent of data-drift. Retraining on visually similar data with little drift results in a narrower spread. With the changing characteristics of videos, it is challenging to efficiently obtain the resource-accuracy profiles for retraining.

Tradeoffs in inference configurations. Inference pipelines also allow for flexibility in their resource demands at the cost of accuracy through configurations to downsize and sample frames [112]. Reducing the resource allocation to inference pipelines increases the processing latency per frame, which then calls for sub-sampling the incoming frames to match the processing rate, that in turn reduces inference accuracy [63]. Prior work has made dramatic advancements in profilers that efficiently obtain the resource-accuracy relationship for *inference configurations* [74]. We use these efficient inference profilers in our solution, and also to ensure that the inference pipelines keep up with analyzing the live video streams.

Illustrative scheduling example

We use an example with 3 GPUs and two video streams, A and B, to show the considerations in scheduling inference and retraining tasks jointly. Each retraining uses data samples accumulated since the *beginning of* the last retraining (referred to as the "retraining window").[†] To simplify

[†]Continuous learning targets retraining windows of tens of seconds to few minutes [186, 141]. We use 120 seconds in this example. Our solution is robust to and works with any given window duration for its decisions (See §2.6).

the example, we assume the scheduler has knowledge of the resource-accuracy profiles, but these are expensive to get in practice (we describe our efficient solution for profiling in §19). Table 2.1 shows the retraining configurations (Cfg1A, Cfg2A, Cgf1B, and Cgf2B), their respective accuracies after the retraining, and GPU cost. The scheduler is responsible for selecting configurations and allocating resources for inference and retraining jobs.

Uniform scheduling: Building upon prior work in cluster schedulers [153, 2] and video analytics systems [63], a baseline solution for resource allocation evenly splits the GPUs between video streams, and each stream evenly partitions its allocated GPUs for retraining and inference tasks; see Figure 2.4a. Just like model training systems [53, 92, 91], the baseline always picks the configuration for retraining that results in the highest accuracy (Cfg1A, Cfg1B for both windows).

Figure 2.4c shows the *inference* accuracies of the two live streams. We see that when the retraining tasks take resources away from the inference tasks, the inference accuracy drops significantly ($65\% \rightarrow 49\%$ for video A and $50\% \rightarrow 37.5\%$ for video B in Window 1). While the inference accuracy increases *after* retraining, it leaves too little time in the window to reap the benefit of retraining. Averaged across both retraining windows, the inference accuracy across the two video streams is only 56% because the gains due to the improved accuracy of the retrained model are undercut by the time taken for retraining (during which inference accuracy suffered).

Accuracy-optimized scheduling: Figures 2.4b and 2.4d illustrate an accuracy-optimized scheduler, which by taking a holistic view on the multi-dimensional tradeoffs, provides an an average inference accuracy of 73%. In fact, to match the accuracies, the above uniform scheduler would require nearly twice the GPUs (i.e., 6 GPUs instead of 3 GPUs).

This scheduler makes three key improvements. First, the scheduler selects the hyperparameter configurations based on their accuracy improvements *relative* to their GPU cost. It selects lower accuracy options (Cfg2A/Cfg2B) instead of the higher accuracy ones (Cfg1A/Cfg1B) because these configurations are substantially cheaper (Table 2.1). Second, the scheduler *prioritizes* retraining tasks that yield higher accuracy, so there is more time to reap the benefit from retraining. For example, the scheduler prioritizes B's retraining in Window 1 as its inference accuracy after retraining increases by 35% (compared to 5% for video A). Third, the scheduler controls the accuracy drops during retraining by balancing the retraining time and the resources taken away from inference.

2.4 Ekya: Solution Description

Continuous training on limited edge resources requires smartly deciding when to retrain each video stream's model, how much resources to allocate, and what configurations to use. Making these decisions presents two challenges.

First, the decision space of multi-dimensional configurations and resource allocations is computationally more complex than two fundamentally challenging problems of multi-dimensional knapsack and multi-armed bandits (§2.4). Hence, we design a **thief scheduler** (§2.4), a heuristic that makes the joint retraining-inference scheduling tractable in practice.

Second, the scheduler requires the model's exact performance (in resource usage and inference accuracy), but this requires retraining using all the configurations. We address this challenge with

our **micro-profiler** (§19), which retrains only a few select configurations on a fraction of the data. Figure 2.5 presents an overview of Ekya's components.

Formulation of joint inference and retraining

The problem of joint inference and retraining aims to maximize overall inference accuracy for all video streams $\mathcal{V}$ in a retraining window T with duration $\|T\|$. All work must be done in $\mathcal{G}$ GPUs. Thus, the total compute capability is $\mathcal{G}\|T\|$ GPU-time. Without loss of generality, let δ be the smallest granularity of GPU allocation. Each video $v \in V$ has a set of *retraining* configurations Γ and a set of *inference* configurations Λ (§2.3). Table 2.4 (§2.11) lists the notations.

Decisions. For each video $v \in \mathcal{V}$ in a window T, we decide: (1) the retraining configuration $\gamma \in \Gamma$ ($\gamma = \emptyset$ means no retraining); (2) the inference configuration $\lambda \in \Lambda$; and (3) how many GPUs (in multiples of δ) to allocate for retraining ($\mathcal{R}$) and inference ($\mathcal{I}$). We use binary variables $\phi_{v\gamma\lambda\mathcal{R}\mathcal{I}} \in \{0, 1\}$ to denote these decisions (see Table 2.4 §2.11 for the definition). These decisions require $C_T(v, \gamma, \lambda)$ GPU-time and yields overall accuracy of $A_T(v, \gamma, \lambda, \mathcal{R}, \mathcal{I})$. $A_T(v, \gamma, \lambda, \mathcal{R}, \mathcal{I})$ is averaged across the window T (§2.3), and the above decisions determine the inference accuracy at *each point in time*.

Optimization. Maximize the inference accuracy averaged across all videos in a retraining window within the GPU limit.

$$\arg\max_{\phi_{v\gamma\lambda\mathcal{R}\mathcal{I}}} \frac{1}{\|\mathcal{V}\|} \sum_{\substack{\forall v\in\mathcal{V},\forall\gamma\in\Gamma,\forall\lambda\in\Lambda,\\ \forall\mathcal{R},\forall\mathcal{I}\in\{0,1,\ldots,\frac{\mathcal{G}}{\delta}\}}} \phi_{v\gamma\lambda\mathcal{R}\mathcal{I}} \cdot A_T(v, \gamma, \lambda, \mathcal{R}, \mathcal{I})$$

subject to

$$\begin{aligned}
&1. \sum_{\substack{\forall v\in\mathcal{V},\forall\gamma\in\Gamma,\forall\lambda\in\Lambda,\\ \forall\mathcal{R},\forall\mathcal{I}}} \phi_{v\gamma\lambda\mathcal{R}\mathcal{I}} \cdot C_T(v, \gamma, \lambda) \leq \mathcal{G}\|T\| \\
&2. \sum_{\substack{\forall v\in\mathcal{V},\forall\gamma\in\Gamma,\forall\lambda\in\Lambda,\\ \forall\mathcal{R},\forall\mathcal{I}}} \phi_{v\gamma\lambda\mathcal{R}\mathcal{I}} \cdot (\mathcal{R} + \mathcal{I}) \leq \frac{\mathcal{G}}{\delta} \\
&3. \sum_{\substack{\forall\gamma\in\Gamma,\forall\lambda\in\Lambda,\\ \forall\mathcal{R},\forall\mathcal{I}}} \phi_{v\gamma\lambda\mathcal{R}\mathcal{I}} \leq 1, \forall v \in \mathcal{V}
\end{aligned} \tag{2.1}$$

The first constraint ensures that the GPU allocation does not exceed the available GPU-time $\mathcal{G}\|T\|$ in the retraining window. The second constraint limits the *instantaneous* allocation (in multiples of δ) to never exceed the available GPUs. The third constraint ensures that at most one configuration is picked for retraining and inference each for a video v.

Our analysis in §2.11 shows that the above optimization problem is *harder* than the multi-dimensional binary knapsack problem and modeling the uncertainty of $A_T(v, \gamma, \lambda, \mathcal{R}, \mathcal{I})$ is more challenging than the multi-armed bandit problem.

Thief Scheduler

Our scheduling heuristic makes the scheduling problem tractable by decoupling resource allocation (i.e., $\mathcal{R}$ and $\mathcal{I}$) and configuration selection (i.e., γ and λ) (Algorithm 1). We refer to Ekya's scheduler as the "thief" scheduler and it iterates among all inference and retraining jobs as follows.

(1) It starts with a fair allocation for all video streams $v \in V$ (line 2 in Algorithm 1). In each step, it iterates over all the inference and retraining jobs of each video stream (lines 5-6), and *steals* a tiny quantum Δ of resources (in multiples of δ; see Table 2.4, §2.11) from each of the other jobs (lines 10-11).

(2) With the new resource allocations (temp_alloc[]), it then selects configurations for the jobs using the PickConfigs method (line 14 and Algorithm 2, §2.11) that iterates over all the configurations for inference and retraining for each video stream. For inference jobs, among all the configurations whose accuracy is $\geq a_{\text{MIN}}$, PickConfigs picks the configuration with the highest accuracy that can keep up with the inference of the live video stream given current allocation (line 3-4, Algorithm 2, §2.11).

For retraining jobs, PickConfigs picks the configuration that maximizes the accuracy $A_T(v, \gamma, \lambda, \mathcal{R}, \mathcal{I})$ over the retraining window for each video v (lines 6-12, Algorithm 2, §2.11). EstimateAccuracy (line 7, Algorithm 2, §2.11) aggregates the instantaneous accuracies over the retraining window for a given pair of inference configuration (chosen above) and retraining configuration. Ekya's micro-profiler (§19) provides the estimate of the accuracy and the time to retrain for a retraining configuration when 100% of GPU is allocated, and EstimateAccuracy proportionately scales the GPU-time for the current allocation (in temp_alloc[]) and training data size. In doing so, it avoids configurations whose retraining durations exceed $\|T\|$ with the current allocation (first constraint in Eq. 2.1).

(3) After reassigning the configurations, Ekya uses the estimated average inference accuracy (accuracy_avg) over the retraining window (line 14 in Algorithm 1) and keeps the new allocations only if it improves up on the accuracy from prior to stealing the resources (line 15 in Algorithm 1).

The thief scheduler repeats the process till the accuracy stops increasing (lines 15-20 in Algorithm 1) and until all the jobs have played the "thief". Algorithm 1 is invoked at the beginning of each retraining window, as well as on the completion of every training job during the window.

Design rationale: We call out the key aspects that makes the scheduler's decision efficient by pruning the search space.

- *Coarse allocations:* The thief scheduler allocates GPU resources in quantums of Δ. Intuitively, Δ is the step size for allocation used by the scheduler. Thus, the final resource allocation from the thief scheduler is within Δ of the optimal allocation. We empirically pick a Δ that is coarse yet accurate enough in practice, while being mindful of modern GPUs[127]; see §2.6. Algorithm 1 ensures that the total allocation is within the limit (second constraint in Eq 2.1).

- *Reallocating resources only when a retraining job completes:* Although one can reallocate GPU resource among jobs at finer temporal granularity (*e.g.,* whenever a retraining job

Algorithm 1: Thief Scheduler.

```
Data: Training (Γ) and inference (Λ) configurations
Result: GPU allocations R and I, chosen configurations (γ ∈ Γ, λ ∈ Λ) ∀v ∈ V
all_jobs[] = Union of inference and training jobs of videos V;
/* Initialize with fair allocation                                  */
best_alloc[] = fair_allocation(all_jobs);
best_configs[], best_accuracy_avg = PickConfigs(best_alloc);
/* Thief resource stealing                                          */
for thief_job in all_jobs[] do
    for victim_job in all_jobs[] do
        if thief_job == victim_job then continue;
        temp_alloc[] ← best_alloc[];
        while true do
            /* Δ is the increment of stealing                       */
            temp_alloc[victim_job] −= Δ;
            temp_alloc[thief_job] += Δ;
            if temp_alloc[victim_job] < 0 then break ;
            /* Calculate accuracy over retraining window and pick
               configurations.                                      */
            temp_configs[], accuracy_avg = PickConfigs(temp_alloc[]);
            if accuracy_avg > best_accuracy_avg then
                best_alloc[] = temp_alloc[];
                best_accuracy_avg = accuracy_avg;
                best_configs[] = temp_configs[];
            else
                break;
return best_alloc[], best_configs[];
```

has reached a high accuracy), we empirically find that the gains from such complexity is marginal.

- *Pruned configuration list:* Our micro-profiler (described next) speeds up the thief scheduler by giving it only the more promising configurations. Thus, the list Γ used in Algorithm 1 is significantly smaller than the exhaustive set.

Performance estimation with micro-profiling

Ekya's scheduling decisions in §2.4 rely on estimations of post-retraining accuracy and resource demand of the retraining configurations. Specifically, at the beginning of each retraining window T, we need to *profile* for each video v and each configuration $\gamma \in \Gamma$, the accuracy after retraining using γ and the corresponding time taken to retrain.

Profiling in Ekya vs. hyperparameter tuning: While Ekya's profiling may look similar to hyperparameter tuning (e.g., [164, 93, 95, 115]) at first blush, there are two key differences. First, Ekya needs the performance estimates of a broad set of candidate configurations for the thief scheduler, not just of the single best configuration, because the best configuration is jointly decided across the many retraining and inference jobs. Second, in contrast to hyperparameter tuning which runs separately of the eventual inference/training, Ekya's profiling must share compute resource with all retraining and inference.

Opportunities: Ekya leverages three empirical observations for efficient profiling of the retraining configurations. (i) Resource demands of the configurations are deterministic. Hence, we measure the GPU-time taken to retrain for *each epoch* in the current retraining window when 100% of the GPU is allocated to the retraining. This GPU-time must then be re-scaled for varying number of epochs, GPU allocations, and training data sizes in Algorithm 1. For re-scaling number of epochs and training data sizes, we linearly scale the GPU-time. For re-scaling GPU allocations, we use an offline computed profile of the model throughput for different resource allocations to account for sub-linear scaling. Our real testbed-based evaluation shows that these rescaling functions works well in practice. (ii) Post-retraining accuracy can be roughly estimated by training on a small subset of training data for a handful of epochs. (iii) The thief scheduler's decisions are not impacted by small errors in the estimations.

Micro-profiling design: The above insights inspired our approach, called *micro-profiling*, where for each video, we test the retraining configurations on a *small subset* of the retraining data and only for a *small number* of epochs (well before models converge). Our micro-profiler is $100\times$ more efficient than exhaustive profiling (of all configurations on the entire training data), while predicting accuracies with an error of 5.8%, which is low enough in practice to *mostly* ensure that the thief scheduler makes the same decisions as it would with a fully accurate prediction. We use these insights to now explain the techniques that make Ekya's micro-profiling efficient.

1) Training data sampling: Ekya's micro-profiling works on only a small fraction (say, $5\% - 10\%$) of the training data in the retraining window (which is already a subset of all the videos accumulated in the retraining window). While we considered weighted sampling techniques for the micro-profiling, we find that uniform random sampling is the most indicative of the configuration's performance on the full training data, since it preserves all the data distributions and variations.

2) Early termination: Similar to data sampling, Ekya's micro-profiling only tests each configuration for a small number (say, 5) of training epochs. Compared to a full fledged profiling that needs few tens of epochs to converge, such early termination greatly speeds up the micro-profiling process.

After early termination on the sampled training data, we obtain the (validation) accuracy of each configuration at each epoch it was trained. We then fit the accuracy-epoch points to the a non-linear curve model from [134] using a non-negative least squares solver [155]. This model is then used to extrapolate the accuracy that would be obtained by retraining with all the data for larger number of epochs. The use of this extrapolation is consistent with similar work in this space [106, 134].

3) Pruning bad configurations: Finally, Ekya's micro-profiling also prunes out those configura-

tions for micro-profiling (and hence, for retraining) that have historically not been useful. These are configurations that are significantly distant from the configurations on the Pareto curve of the resource-accuracy profile (see Figure 2.3b), and thus unlikely to be picked by the thief scheduler. To bootstrap pruning, all configurations are evaluated in the first window. After every 2 windows, a fixed fraction of the worst performing configurations are dropped. While first few retraining windows must explore a big space of configurations, the search space size drops exponentially over time. Avoiding these configurations improves the efficiency of the micro-profiling.

Annotating training data: For both the micro-profiling as well as the retraining, `Ekya` acquires labels using a "golden model" (§2.2). This is a high-cost but high-accuracy model trained on a large dataset. As explained in §2.2, the golden model cannot keep up with inference on the live videos and we use it to label only a small subset of the videos for retraining. The delay of annotating training data with the golden model is accounted by the scheduler as follows: we subtract the data annotation delay from the retraining window and only pass the remaining time of the window to Algorithm 2 (§2.11).

2.5 Implementation and Experimental Setup

Implementation: `Ekya` uses PyTorch [130] for running and training ML models, and each component is implemented as a collection of long-running processes with the Ray[117] actor model. The micro-profiler and training/inference jobs run as independent actors which are controlled by the thief scheduler actor. `Ekya` achieves fine-grained and dynamic reallocation of GPU between training and inference processes using Nvidia MPS [127], which provides resource isolation within a GPU by intercepting CUDA calls and rescheduling them. Our implementation also adapts to errors in profiling by reactively adjusting its allocations if the actual model performance diverges from the predictions of the micro-profiler. `Ekya`'s code and datasets are available at the project page: aka.ms/ekya

Datasets: We use both on-road videos captured by dashboard cameras as well as urban videos captured by mounted cameras. The dashboard camera videos are from cars driving through cities in the US and Europe, Waymo Open [133] (1000 video segments with in total 200K frames) and Cityscapes [110] (5K frames captured by 27 cameras) videos. The urban videos are from stationary cameras mounted in a building ("Urban Building") as well as from five traffic intersections ("Urban Traffic"), both collected over 24-hour durations. We use a retraining window of 200 seconds in our experiments, and split each of the videos into 200 second segments. Since the Waymo and Cityscapes dataset do not contain continuous timestamps, we create retraining windows by concatenating images from the same camera in chronological order to form a long video stream and split it into 200 second segments.

DNNs: We demonstrate `Ekya`'s effectiveness on two machine learning tasks – object classification and object detection – using multiple compressed edge DNNs for each task: (i) object classification using ResNet18[77], MobileNetV2[102] and ShuffleNet[198], and (ii) object detection using TinyYOLOv3[145] and SSD[98]. As explained in §2.2, we use an expensive golden

model (ResNeXt 101 [182] for object classification and YOLOv3 [145] for object detection) to get ground truth labels for training and testing.

Testbed and trace-driven simulator: We run Ekya's implementation on AWS EC2 p3.2xlarge instances for 1 GPU experiments and p3.8xlarge for 2 GPU experiments. Each instance has Nvidia V100 GPUs with NVLink interconnects.

We also built a simulator to test Ekya under a wide range of resource constraints, workloads, and longer durations. The simulator takes as input the accuracy and resource usage (in GPU time) of training/inference configurations logged from our testbed. For each training job, we log the accuracy over GPU-time. We also log the inference accuracy on the real videos. This exhaustive trace allows us to mimic the jobs with high fidelity under different scheduling policies.

Retraining configurations: Our retraining configurations combine the number of epochs to train, batch size, number of neurons in the last layer, number of layers to retrain, and the fraction of data between retraining windows to use for retraining (§2.3). For the object detection models (TinyYOLO and SSDLite), we set the batch size to 8 and the fraction of layers frozen between 0.7 and 0.9. The resource requirements of the configurations for the detection models vary by $153\times$.

Baselines: Our baseline, called *uniform scheduler*, uses (a) a fixed retraining configuration, and (b) a static retraining/inference resource allocation (these are adopted by prior schedulers [153, 2, 63]). For each dataset, we test all retraining configurations on a hold-out dataset [‡] (*i.e.,* two video streams that were never used in later tests) to produce the Pareto frontier of the accuracy-resource tradeoffs (*e.g.,* Figure 2.3). The uniform scheduler then picks two points on the Pareto frontier as the fixed retraining configurations to represent "high" (Config 1) and "low" (Config 2) resource usage, and uses one of them for all retraining windows in a test.

We also consider two alternatives in §2.6. (1) *offloading retraining to the cloud*, and (2) *caching and re-using a retrained model* from history based on various similarity metrics.

2.6 Evaluation

We evaluate Ekya's performance, and the key findings are:

1) Compared to static retraining baselines, Ekya achieves upto 29% higher accuracy for compressed vision models in both classification and detection. For the baseline to match Ekya's accuracy, it would need $4\times$ additional GPU resources. (§2.6)

2) Both micro-profiling and thief scheduler contribute sizably to Ekya's gains. (§2.6) In particular, the micro-profiler estimates accuracy with low median errors of 5.8%. (§2.6)

3) The thief scheduler efficiently makes its decisions in 9.4s when deciding for 10 video streams across 8 GPUs with 18 configurations per model for a 200s retraining window. (§2.6)

4) Compared to alternate designs, including reusing cached history models trained on similar data/scenarios as well as retraining the models in the cloud, Ekya achieves significantly higher accuracy without the network costs (§2.6).

[‡]The same hold-out dataset is used to customize the off-the-shelf DNN inference model. This is a common strategy in prior work (*e.g.,* [34]).

Scheduler	Capacity		Scaling factor
	1 GPU	2 GPUs	
Ekya	**2**	**8**	**4x**
Uniform (Config 1, 50%)	2	2	1x
Uniform (Config 2, 90%)	2	4	2x
Uniform (Config 2, 50%)	2	4	2x
Uniform (Config 2, 30%)	0	2	-

Table 2.2: Capacity (number of video streams that can be concurrently supported subject to accuracy target 0.75) vs. number of provisioned GPUs. `Ekya` **scales better than the uniform baselines with more available compute resource.**

Overall improvements

We evaluate `Ekya` and the baselines along three dimensions— *inference accuracy* (% of images correctly classified for object classification, F1 score (measured at a 0.3 threshold for the Intersection-over-Union of the bounding box) for detection), *resource consumption* (in GPU time), and *capacity* (the number of concurrently processed video streams). Note that the evaluation always keeps up with the video frame rate (*i.e.,* no indefinite frame queueing). By default we evaluate the performance of Ekya on ResNet18 models, but we also show that it generalizes to other model types and vision tasks.

Accuracy vs. Number of concurrent video streams: Figure 2.6 shows the ResNet18 model's accuracy with `Ekya` and the baselines when analyzing a growing number of concurrent video streams under a fixed number of provisioned GPUs for Waymo and Cityscapes datasets. The uniform baselines use different combinations of pre-determined retraining configurations and resource partitionings. For consistency, the video streams are shuffled and assigned an id (0-10), and are then introduced in the same increasing order of id in all experiments. This ensures that different schedulers tested for k parallel streams use the same k streams, and these k streams are always a part of any k' streams ($k' > k$) used for testing.

As the number of video streams increases, `Ekya` enjoys a growing advantage (upto 29% under 1 GPU and 23% under 2 GPU) in accuracy over the uniform baselines. This is because `Ekya` gradually shifts more resource from retraining to inference and uses cheaper retraining configurations. In contrast, increasing the number of streams forces the uniform baseline to allocate less GPU cycles to each inference job, while retraining jobs, which use fixed configurations, slow down and take the bulk of each window.

Generalizing to other ML models: `Ekya`'s thief scheduler can be readily applied to any ML model and task (e.g., classification or detection) that needs to be fine-tuned continuously on newer data. To demonstrate this, we evaluate `Ekya` with:

- *Other object classifiers:* Figure 2.7a shows the performance of `Ekya` when running MobileNetV2 and ShuffleNet as the edge models in two independent setups for object classi-

fication at the edge. Continuing the trend that we observed for ResNet18 (in Figure 2.6), Figure 2.7a shows that Ekya leads to up to 22% better accuracy than uniform baselines.

- *Object detection models:* In addition to object classification, we also evaluate using object detection tasks which detect the bounding boxes of objects in the video stream. Figure 2.7b shows Ekya outperforms the uniform baseline's F1 score by 19% when processing same number of concurrent video streams. Importantly, Ekya's design broadly applies to new tasks without any systemic changes.

These gains stem from Ekya's ability to navigate the rich resource-accuracy space of models by carefully selecting training and inference hyperparameters (e.g., the width multiplier in MobileNetV2, convolution sparsity in ShuffleNet). For the rest of our evaluation, we only present results with ResNet18 though the observations hold for other models.

Number of video streams vs. provisioned resource: We compare Ekya's *capacity* (defined by the maximum number of concurrent video streams subject to an accuracy threshold) with that of uniform baseline, as more GPUs are available. Setting an accuracy threshold is common in practice, since applications usually require accuracy to be above a threshold for the inference to be usable. Table 2.2 uses the Cityscapes results (Figure 2.6) to derive the scaling factor of capacity vs. the number of provisioned GPUs and shows that with more provisioned GPUs, Ekya scales faster than uniform baselines.

Accuracy vs. provisioned resource: Finally, Figure 2.8 stress-tests Ekya and the uniform baselines to process 10 concurrent video streams and shows their average inference accuracy under different number of GPUs. To scale to more GPUs, we use the simulator (§2.5), which uses profiles recorded from real tests and we verified that it produced similar results as the implementation at small-scale. As we increase the number of provisioned GPUs, we see that Ekya consistently outperforms the best of the two baselines by a considerable margin and more importantly, with 4 GPUs Ekya achieves higher accuracy (marked with the dotted horizontal line) than the baselines at 16 GPUs (*i.e.,* 4× resource saving).

The above results show that Ekya is more beneficial when there is high contention for the GPU on the edge. Under low contention, the room for improvement shrinks. Contention is, however, common in the edge since the resources are tightly provisioned to minimize their idling.

Understanding Ekya's improvements

Resource allocation across streams: Figure 2.9 shows Ekya's resource allocation across two example video streams over several retraining windows. In contrast to the uniform baselines that use the same retraining configuration and allocate equal resource to retraining and inference (when retraining takes place), Ekya retrains the model only when it benefits and allocates different amounts of GPUs to the retraining jobs of video streams, depending on how much accuracy gain is expected from retraining on each stream. In this case, more resource is diverted to video stream #1 (#1 can benefit more from retraining than #2) and both video streams achieve much higher accuracies (0.82 and 0.83) than the uniform baseline.

Component-wise contribution: Figure 2.10a understands the contributions of resource allocation and configuration selection (on 10 video streams with 4 GPUs provisioned). We construct two variants from Ekya: *Ekya-FixedRes*, which removes the smart resource allocation in Ekya (*i.e.,* using the inference/training resource partition of the uniform baseline), and *Ekya-FixedConfig* removes the microprofiling-based configuration selection in Ekya (*i.e.,* using the fixed configuration of the uniform baseline). Figure 2.10a shows that both adaptive resource allocation and configuration selection has a substantial contribution to Ekya's gains in accuracy, especially when constrained (i.e., fewer resources are provisioned).

Retraining window sensitivity analysis: Figure 2.10b evaluates the sensitivity of Ekya to the retraining window size. Ekya is robust to different retraining window sizes. When the retraining window size is too small (10 seconds), the accuracy of Ekya is equivalent to no retraining accuracy due to insufficient time and resources for retraining. As the window increases, Ekya's performance quickly ramps up because the thief scheduler is able to allocate resources to retraining. As the retraining window size further increases Ekya's performance slowly starts moderately degrading because of the inherent limitation in capacity of compressed models (§2.2).

Impact of scheduling granularity: A key parameter in Ekya's scheduling algorithm (§2.4) is the allocation quantum Δ: it controls the runtime of the scheduling algorithm and the granularity of resource allocation. In our sensitivity analysis with 10 video streams, we see that increasing Δ from 1.0 (coarse-grained; one full GPU) to 0.1 (fine-grained; fraction of a GPU), increases the accuracy substantially by $\sim 8\%$. Though the runtime also increases to 9.5 seconds, it is still a tiny fraction (4.7%) of the retraining window (200s).

Effectiveness of micro-profiling

The absolute cost of micro-profiling is small; for our experiments, micro-profiling takes 4.4 seconds for a 200s window.

Errors of microprofiled accuracy estimates: Ekya's micro-profiler estimates the accuracy of each configuration (§19) by training it on a subset of the data for a small number of epochs. To evaluate the micro-profiler's estimates, we run it on all configurations for 5 epochs and on 10% of the retraining data from all streams of the Cityscapes dataset, and calculate the estimation error against the retrained accuracies when trained on 100% of the data for 5, 15 and 30 epochs. Figure 2.11a plots the distribution of the errors in accuracy estimation and and show that the micro-profiled estimates are largely unbiased with an median absolute error of 5.8%.

Sensitivity to microprofiling estimation errors: Finally, we test the impact of accuracy estimation errors (§19) on Ekya. We add gaussian noise on top of the predicted retraining accuracy when the microprofiler is queried. Figure 2.11b shows that Ekya is robust to accuracy estimate errors: with upto 20% error (which covers all errors in Figure 2.11a) in the profiler prediction, the maximum accuracy drop is 3%.

	Bandwidth (Mbps)		Acc.	Bandwidth Gap	
	Uplink	Downlink		Uplink	Downlink
Cellular	5.1	17.5	68.5%	10.2×	3.8×
Satellite	8.5	15	69.2%	5.9×	4.4×
Cellular (2×)	10.2	35	71.2%	5.1×	1.9×
Ekya	-	-	**77.8%**	**-**	**-**

Table 2.3: Retraining in the cloud under different networks [111, 128, 156] versus using Ekya at the edge. Ekya achieves better accuracy without using expensive satellite and cellular links.

Comparison with alternative designs

Ekya vs. Cloud-based retraining: One may upload a sub-sampled video stream to the cloud, retrain the model, and download the model back to the edge [84]. While this solution is not an option for many deployments due to legal and privacy stipulations [168, 4], we still evaluate this option as it lets the edge servers focus on inference. Cloud-based solutions, however, results in lower accuracy due to significant network delays on the constrained networks typical of edges [156].

For example, consider 8 video streams running ResNet18 and a retraining window of 400 seconds. A HD (720p) video stream at 4Mbps and 10% data sub-sampling (typical in our experiments) amounts to 160Mb of training data per camera per window. Uploading 160Mb for each of the 8 cameras over a 4G uplink (5.1 Mbps [128]) and downloading the trained ResNet18 models (398 Mb each [174]) over the 17.5 Mbps downlink [128] takes 432 seconds (even excluding the model retraining time), which already exceeds the retraining window.

To test on the Cityscapes dataset, we extend our simulator (§2.5) to account for network delays during retraining, and test with 8 videos and 4 GPUs. We use the conservative assumption that retraining in the cloud is "instantaneous" (cloud GPUs are powerful than edge GPUs). Table 2.3 lists the accuracies with cellular 4G links (both one and two subscriptions to meet the 400s retraining window) and a satellite link, which are both indicative of edge deployments [156].

For the cloud alternatives to match Ekya's accuracy, we will need to provision additional uplink capacity of 5×-10× and downlink capacity of 2×-4× (of the already expensive links). In summary, Ekya's edge-based solution is better than a cloud alternate for retraining in *both* accuracy and network usage (Ekya sends no data out of the edge), all while providing privacy for the videos. However, when the edge-cloud network has sufficient bandwidth, e.g., in an enterprise that is provisioned with a private leased connection, then using the cloud to retrain the models can be a viable design choice.

Ekya vs. Re-using pretrained models: An alternative to continuous retraining is to cache *pre-trained* models and reuse them. We pre-train and cache a few tens of DNNs from earlier windows of the Waymo dataset and test four heuristics for selecting cached models. *Class-distribution*-based selection picks the cached DNN whose training data class distribution has the closest Euclidean distance with the current window's data. *Time-of-day*-based selection picks the cached DNN whose training data time matches the current window. *Object-count*-based selection picks

the cached DNN based on similar count of objects. *Location*-based selection picks the cached DNNs trained on the same city as the current window.

Figure 2.12a highlights the advantages of Ekya over different model selection schemes. We find that since time-of-day-based, object-count-based, and location-based model selection techniques are agnostic to the class distributions of training data of cached models, the selected cached models sometimes do not cover all classes in the current window. Even if we take class distribution into account when picking cached models, there are still substantial discrepancies in the appearances of objects between the current window and the history training data. For instance, object appearance can vary due to pose variations, occlusion or different lighting conditions. In Window 3 (Figure 2.12a), not only are certain classes underrepresented in the training data, but the lighting conditions are also adverse. Figure 2.12b presents a box plot of the accuracy difference between Ekya and model selection schemes, where the edges of the box represent the first and third quartiles, the waist is the median, the whiskers represent the maximum and minimum values and the circles represent any outliers. Ekya's continuous retraining of models is robust to scene specific data-drifts and achieves upto 26% higher mean accuracy.

2.7 Limitations and Discussion

Edge hierarchy with heterogeneous hardware. While Ekya's allocates GPU resources on a single edge, in practice, deployments typically consist of a *hierarchy* of edge devices [26]. For instance, 5G settings include an on-premise edge cluster, followed by edge compute at cellular towers, and then in the core network of the operator. The compute resources, hardware (e.g., GPUs, Intel VPUs [14], and CPUs) and network bandwidths change along the hierarchy. Thus, Ekya will have to be extended along two aspects: (a) multi-resource allocation to include both compute and the network in the edge hierarchy; and (b) heterogeneity in edge hardware.

Privacy of video data. As explained in §2.2, privacy of videos is important in real-world deployments, and Ekya's decision to retrain only on the edge device is well-suited to achieving privacy. However, when we extend Ekya to a hierarchy of edge clusters, care has to be taken to decide the portions of the retraining that can happen on edge devices that are *not* owned by the enterprise. Balancing the need for privacy with resource efficiency is a subject for future work.

Generality beyond vision workloads. Ekya's thief scheduler is generally applicable to DNN models since it only requires that the resource-accuracy function be strictly increasing wherein allocation of more resources to training results in increasing accuracy. This property holds true for *most* workloads (vision and language DNNs). However, when this property does *not* hold, further work is needed to prevent Ekya's microprofiler from making erroneous estimations and its thief scheduler from making sub-optimal allocations.

2.8 Related Work

1) ML training systems. For large scale scheduling of training in the cloud, model and data parallel frameworks [36, 99, 3, 120] and various resource schedulers [107, 188, 60, 135, 57, 197] have been developed. These systems, however, target different objectives than `Ekya`, like maximizing parallelism, fairness, or minimizing average job completion. Collaborative training systems [19, 100] work on decentralized data on mobile phones. They focus on coordinating the training between edge and the cloud, and not on training alongside inference.
2) Video processing systems. Prior work has built low-cost, high-accuracy and scalable video processing systems for the edge and cloud [63, 74, 34]. VideoStorm investigates quality-lag requirements in video queries [63]. NoScope exploits difference detectors and cascaded models to speedup queries [34]. Focus uses low-cost models to index videos [67]. Chameleon exploits correlations in camera content to amortize *profiling costs* [74]. Reducto [94] and DDS [41] seek to reduce edge-to-cloud traffic by intelligent frame sampling and video encoding. All of these works optimize only the inference accuracy or the system/network costs of DNN inference, unlike `Ekya`'s focus on retraining. More recently, LiveNAS[85] deploys continuous retraining to update video upscaling models, but focuses on efficiently allocating client-server bandwidth to different subsamples of a single video stream. Instead, Ekya focuses on GPU allocation for maximizing retrained accuracy across multiple video streams.
3) Hyper-parameter optimization. Efficient exploration of hyper-parameters is crucial in training systems to find the model with the best accuracy. Techniques range from simple grid or random search [18], to more sophisticated approaches using random forests [68], Bayesian optimization [164, 169], probabilistic modelling [140], or non-stochastic infinite-armed bandits [93]. Unlike the focus of these techniques on finding the hyper-parameters with the highest accuracy, our focus is on resource allocation. Further, we are focused on the inference accuracy over the retrained window, where producing the best retrained model often turns out to be sub-optimal.
4) Continuous learning. Machine learning literature on continuous learning adapts models as new data comes in. A common approach used is transfer learning [144, 100, 141, 66]. Research has also been conducted on handling catastrophic forgetting [90, 151], using limited amount of training data [170, 142], and dealing with class imbalance [16, 185]. `Ekya` builds atop continuous learning techniques for its scheduling and implementation, to enable them in edge deployments.

2.9 Acknowledgements

We thank the NSDI reviewers and our shepherd, Minlan Yu, for their invaluable feedback. This research is partly supported by NSF (CCF-1730628, CNS-1901466), UChicago CERES Center, a Google Faculty Research Award and gifts from Amazon, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

Notation	Description
$\mathcal{V}$	Set of video streams
v	A video stream ($v \in \mathcal{V}$)
T	A retraining window with duration $\|T\|$
Γ	Set of all retraining configurations
γ	A retraining configuration ($\gamma \in \Gamma$)
Λ	Set of all inference configurations
λ	An inference configuration ($\lambda \in \Lambda$)
$\mathcal{G}$	Total number of GPUs
δ	The unit for GPU resource allocation
$A_T(v, \gamma, \lambda, \mathcal{R}, \mathcal{I})$	Inference accuracy for video v for given configurations and allocations
$C_T(v, \gamma, \lambda)$	Compute cost in GPU-time for video v for given configurations and allocations
$\phi_{v\gamma\lambda\mathcal{R}\mathcal{I}}$	A set of binary variables ($\phi_{v\gamma\lambda\mathcal{R}\mathcal{I}} \in \{0, 1\}$). $\phi_{v\gamma\lambda\mathcal{R}\mathcal{I}} = 1$ iff we use retraining config γ, inference config λ, $\mathcal{R}\delta$ GPUs for retraining, $\mathcal{I}\delta$ GPUs for inference for video v

Table 2.4: Notations used in `Ekya`'s description.

2.10 Appendix

2.11 Thief Scheduler

Romil: Move to rest of the paper

Complexity Analysis.

Assuming all the $A_T(v, \gamma, \lambda, \mathcal{R}, \mathcal{I})$ values are known, the above optimization problem can be reduced to a multi-dimensional binary knapsack problem, a NP-hard problem [105]. Specifically, the optimization problem is to pick binary options ($\phi_{v\gamma\lambda\mathcal{R}\mathcal{I}}$) to maximize overall accuracy while satisfying two capacity constraints (the first and second constraints in Eq 2.1). In practice, however, getting all the $A_T(v, \gamma, \lambda, \mathcal{R}, \mathcal{I})$ is *infeasible* because this requires training the edge DNN using all retraining configurations and running inference using all the retrained DNNs with all possible GPU allocations and inference configurations.

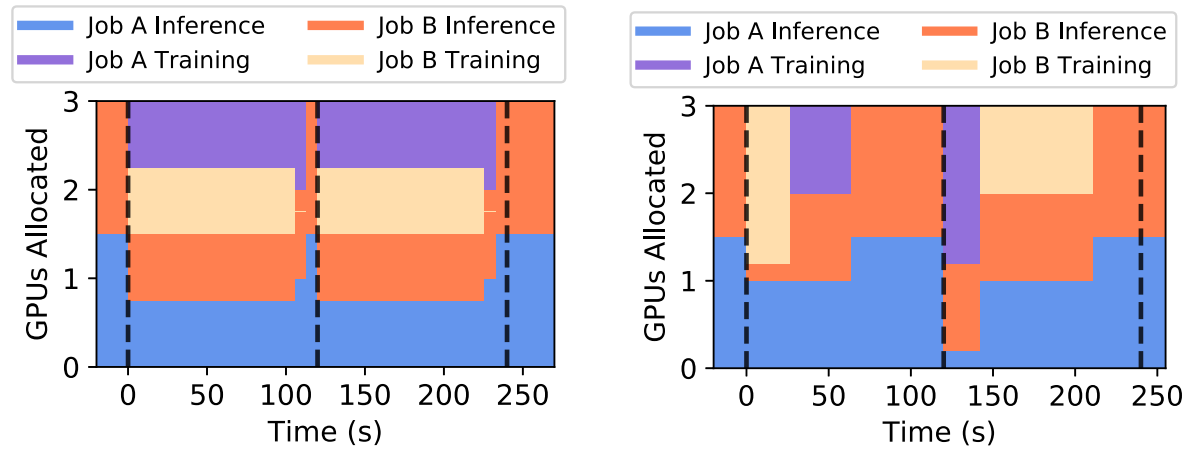
The uncertainty of $A_T(v, \gamma, \lambda, \mathcal{R}, \mathcal{I})$ resembles the multi-armed bandits (MAB) problem [150] to maximize the expected rewards given a limited number of trials for a set of options. Our optimization problem is more challenging than MAB for two reasons. First, unlike the MAB problem, the cost of trials ($C_T(v, \gamma, \lambda)$) varies significantly, and the optimal solution may need to choose cheaper yet less rewarding options to maximize the overall accuracy. Second, getting the reward $A_T(v, \gamma, \lambda, \mathcal{R}, \mathcal{I})$ after each trial requires "ground truth" labels that are obtained using the large golden model, which can only be used judiciously on resource-scarce edges (§2.2).

Algorithm 2: PickConfigs

Data: Resource allocations in temp_alloc[], configurations (Γ and Λ), retraining window T, videos V
Result: Chosen configs $\forall v \in V$, average accuracy over T

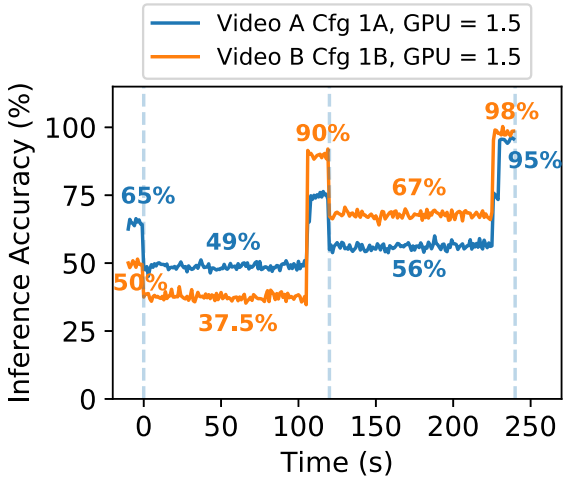
```
chosen_accuracies[] ←{}; chosen_configs[] ←{};
for v in V[] do
    infer_config_pool[] = Λ.where(resource_cost < temp_alloc[v.inference_job] && accuracy
     ≥ a_MIN );
    infer_config = max(infer_config_pool, key=accuracy);
    best_accuracy = 0;
    for train_config in Γ do
        /* Estimate accuracy of inference/training config pair over
           retraining window                                          */
        accuracy = EstimateAccuracy(train_config, infer_config, temp_alloc[v.training_job], T);
        if accuracy > best_accuracy then
            best_accuracy = accuracy;
            best_train_config = train_config;
    chosen_accuracies[v] = best_accuracy;
    chosen_configs[v] = {infer_config, best_train_config};
return chosen_configs[], mean(chosen_accuracies[]);
```

In summary, our optimization problem is computationally more complex than two fundamentally challenging problems (multi-dimensional knapsack and multi-armed bandits).

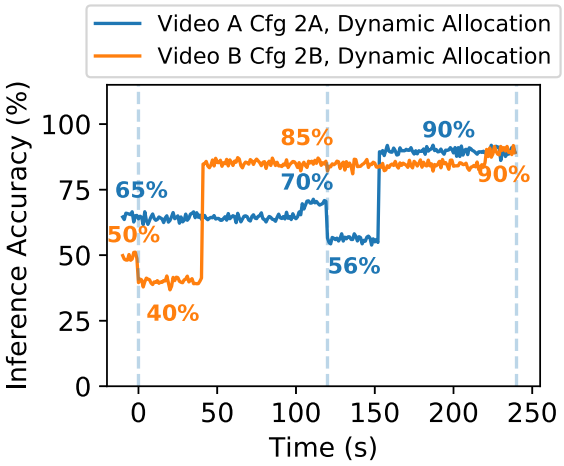


(a) Uniform scheduler

(b) Accuracy-optimal sched.



(c) Uniform scheduler



(d) Accuracy-optimal sched.

Figure 2.4: Resource allocations (top) and inference accuracies (bottom) over time for two retraining windows (each of 120s). The left figures show a uniform scheduler which evenly splits the 3 GPUs, and picks configurations resulting in the most accurate models. The right figures show the accuracy-optimized scheduler that prioritizes resources and optimizes for inference accuracy averaged over the retraining window (73% compared to the uniform scheduler's 56%). The accuracy-optimized scheduler also ensures that inference accuracy never drops below a minimum (set to 40% in this example, denoted as a_{MIN}).

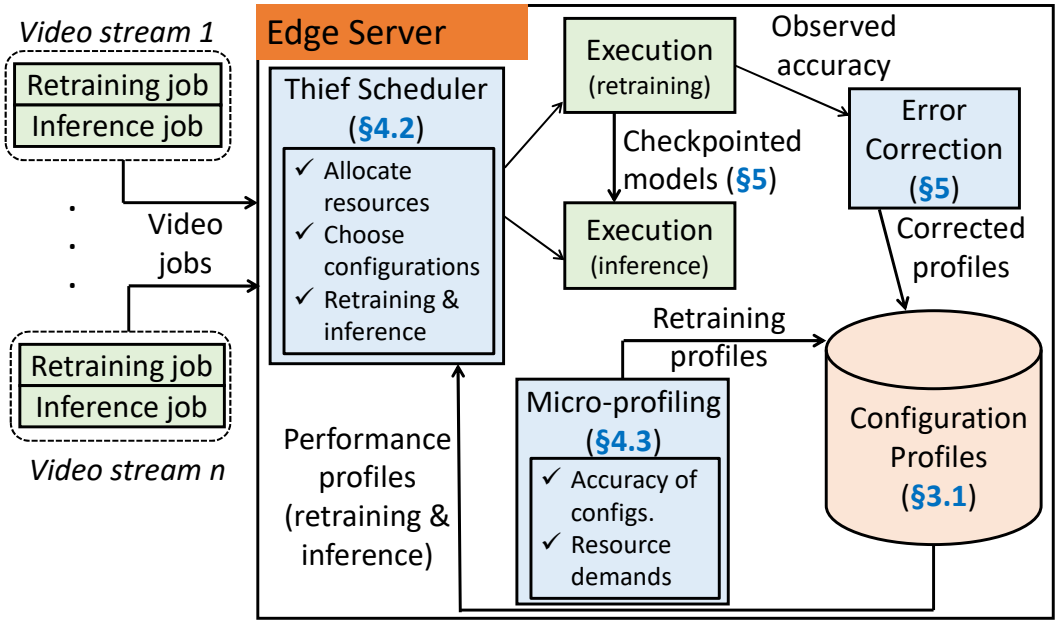
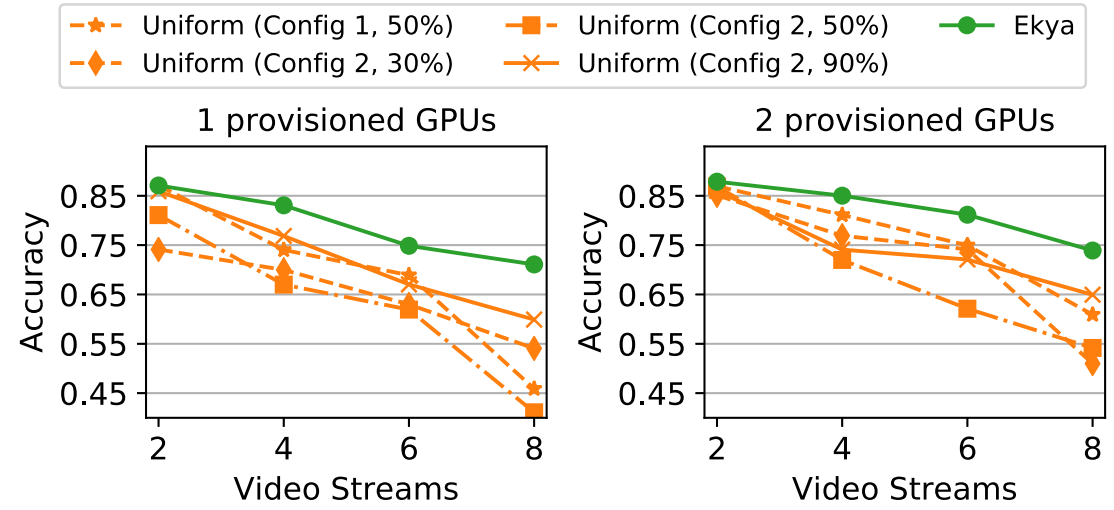
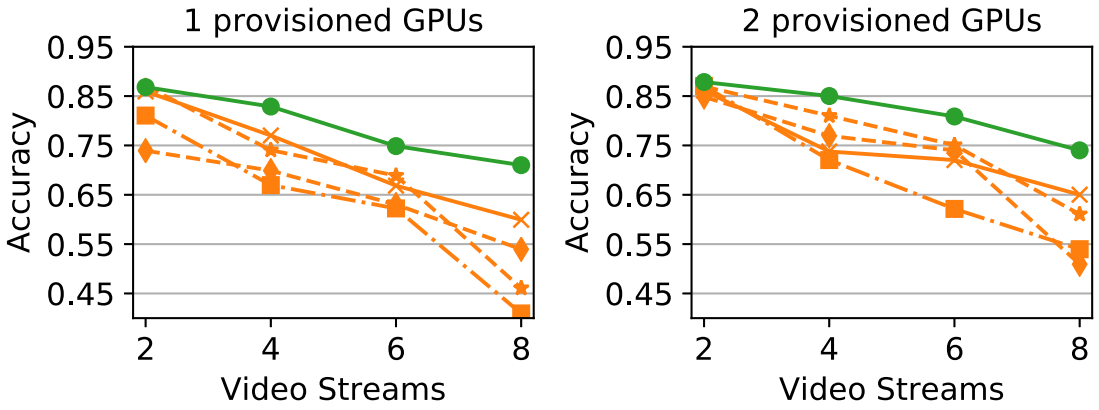


Figure 2.5: `Ekya`**'s components and their interactions.**

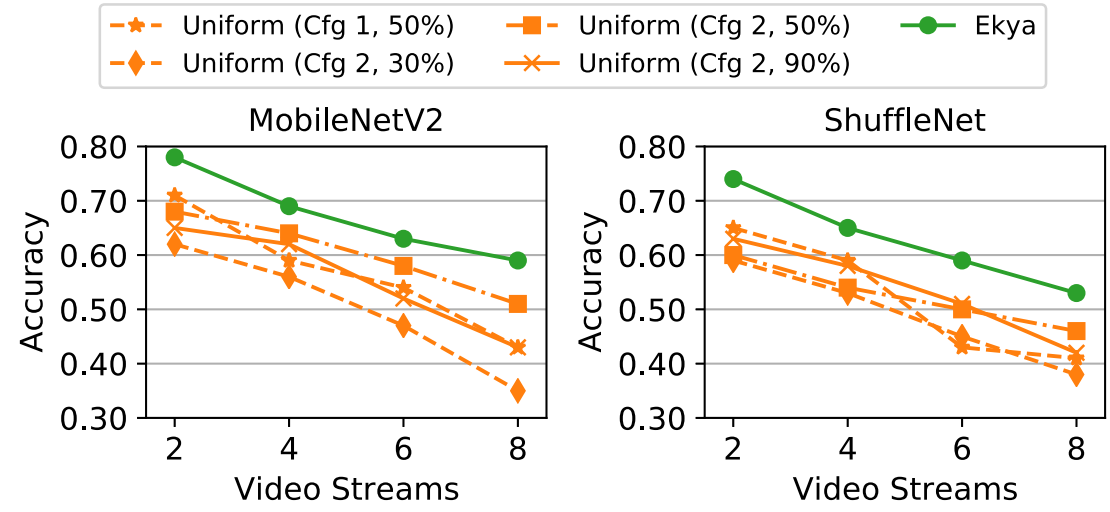


(a) Cityscapes

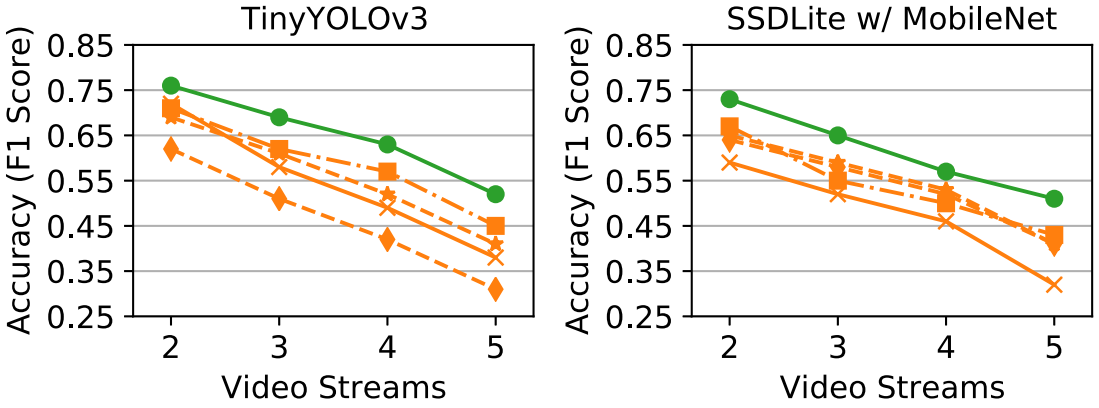


(b) Waymo

Figure 2.6: Effect of adding video streams on accuracy with different schedulers. When more video streams share resources, `Ekya`'s accuracy gracefully degrades while the baselines' accuracy drops faster. ("Uniform (Cfg 1, 90%)" means the uniform scheduler allocates 90% GPU to inference, 10% to retraining)

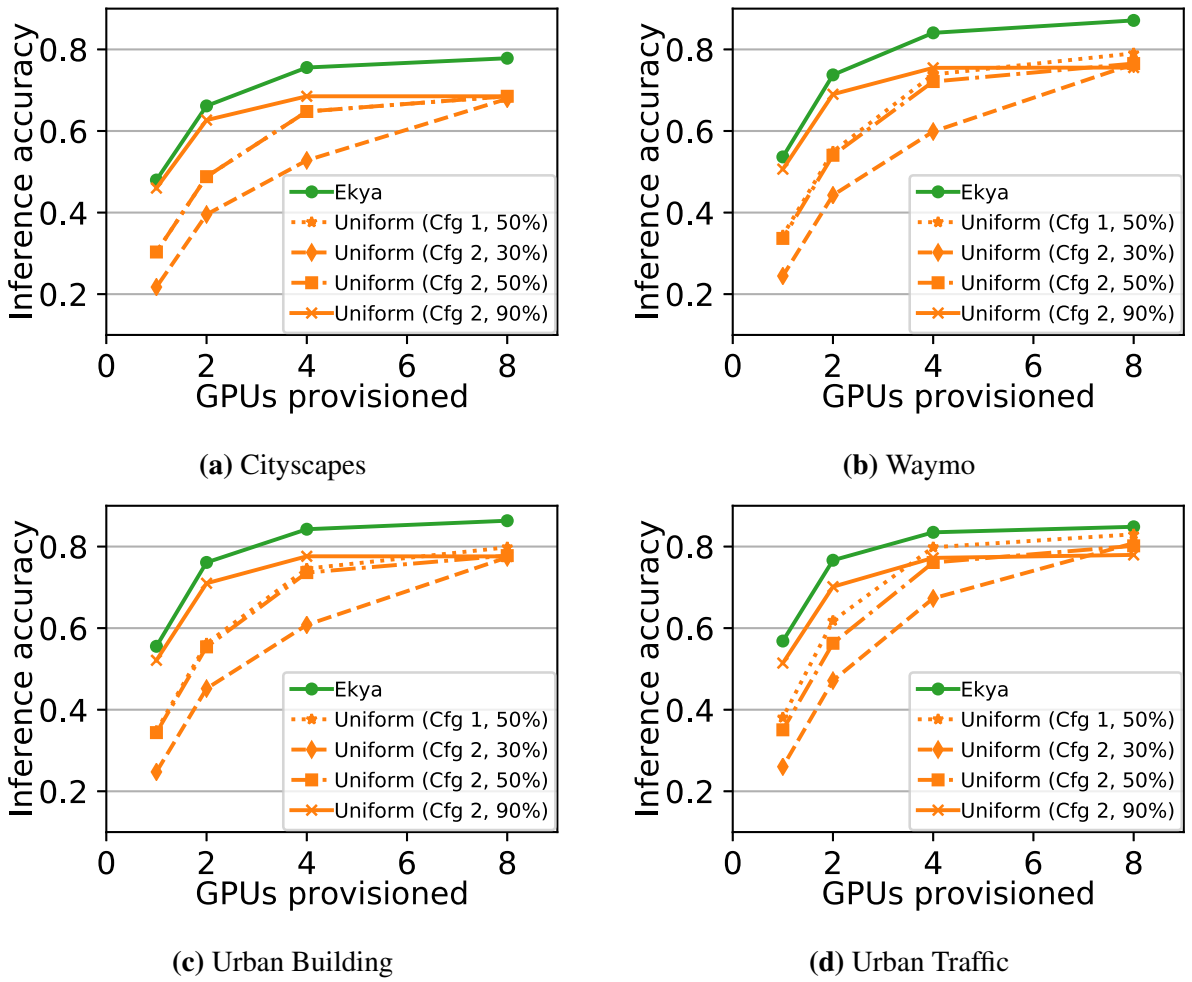


(a) Generalize across object classification models



(b) Object Detection Models

Figure 2.7: Improvement of `Ekya` extends to two more compressed DNN classifiers and two popular object detectors.



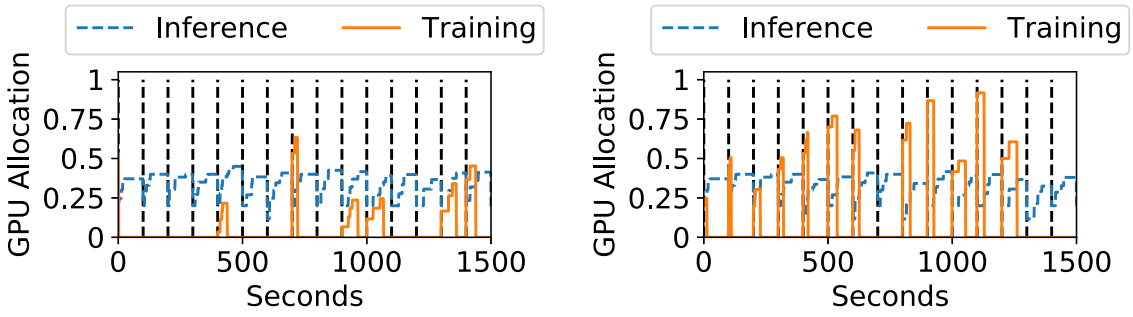
(a) Cityscapes

(b) Waymo

(c) Urban Building

(d) Urban Traffic

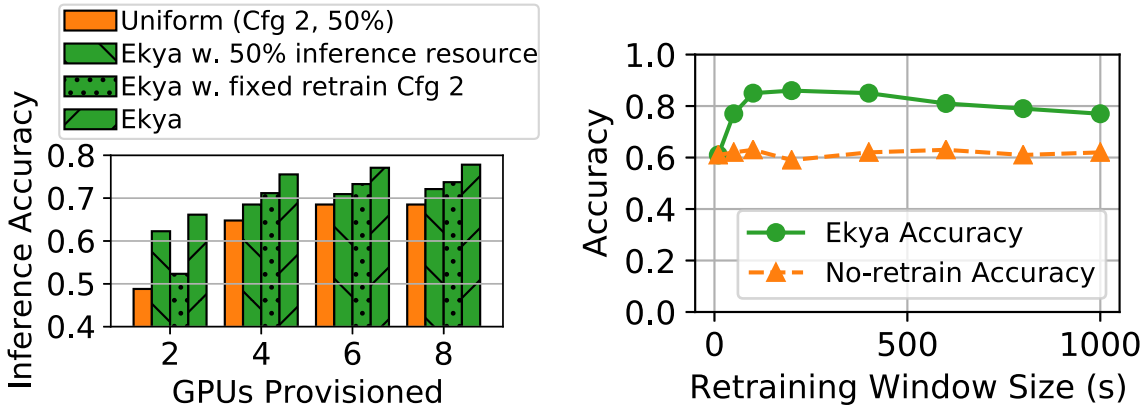
Figure 2.8: Inference accuracy of different schedulers when processing 10 video streams under varying GPU provisionings.



(a) Video stream #1
(Inference accuracy = 0.82)

(b) Video stream #2
(Inference accuracy = 0.83)

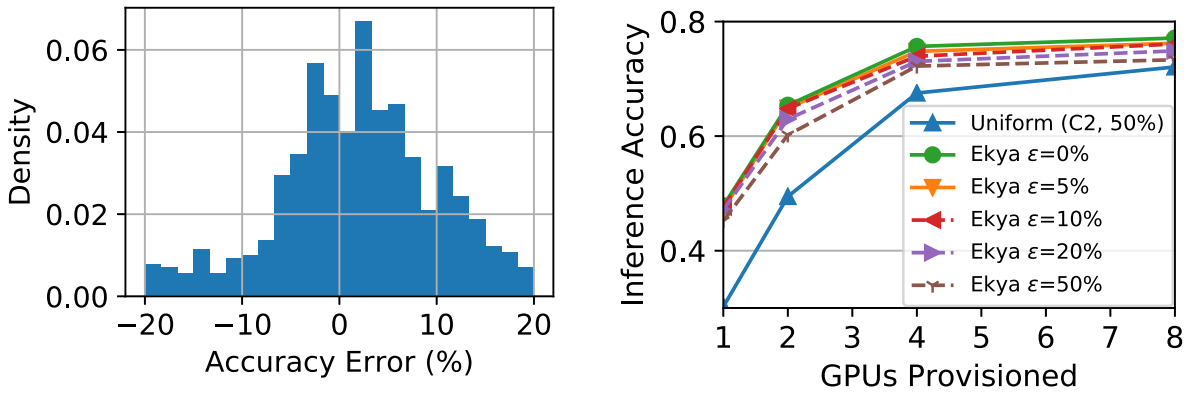
Figure 2.9: `Ekya`'s resource allocation to two video streams over time. `Ekya` adapts when to retrain each stream's model and allocates resource based on the retraining benefit to each stream.



(a) Factor analysis

(b) Sensitivity to retraining window size.

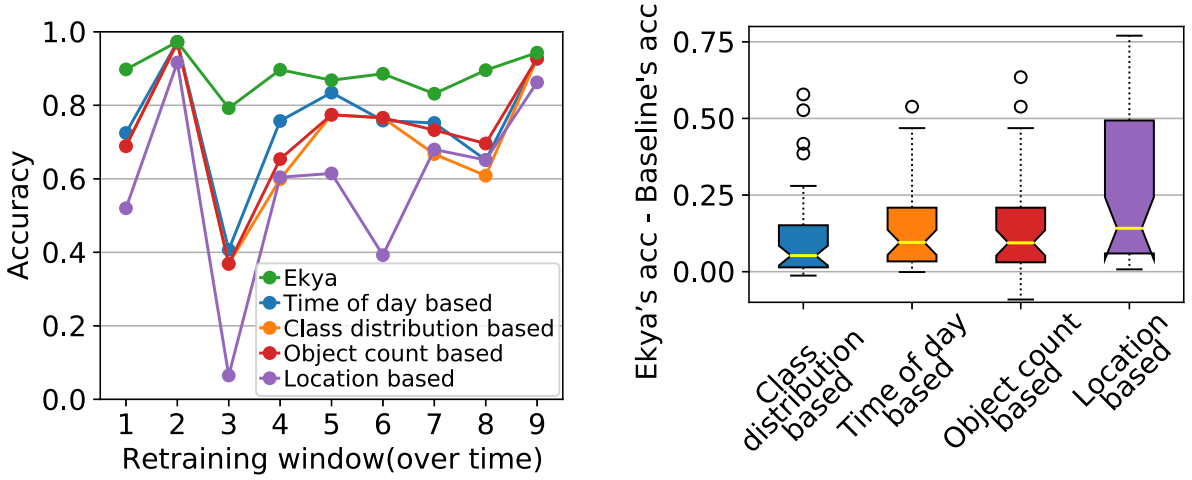
Figure 2.10: (a) Component-wise impact of removing dynamic resource allocation (50% allocation) or removing retraining configuration adaptation (fixed Cfg 2). (b) Robustness of `Ekya` to a wide range of retraining window values.



(a) Distribution of accuracy estimation errors.

(b) Impact of an controlled error ϵ to accuracy estimates.

Figure 2.11: Evaluation of microprofiling performance. (a) shows the distribution of microprofiling's actual estimation errors, and (b) shows the robustness of `Ekya`'s performance against microprofiling's estimation errors.



(a) `Ekya` vs. re-using cached models over time

(b) Average gains in accuracy across video streams

Figure 2.12: `Ekya` vs. re-using cached models. Compared to cached-model selection techniques, models retrained with `Ekya` maintain a consistently high accuracy, since it fully leverages the latest training data and is thus more robust to data-drift.

Chapter 3

Cilantro: Cluster Resource Allocation with Online-Learning

Traditional systems for allocating finite cluster resources among competing jobs have either aimed at providing fairness, relied on users to specify their resource requirements, or have estimated these requirements via surrogate metrics (e.g. CPU utilization). These approaches do not account for a job's real world performance (e.g. P95 latency). Existing performance-aware systems use offline profiled data and/or are designed for specific allocation objectives. In this work, we argue that resource allocation systems should directly account for real-world performance and the varied allocation objectives of users. In this pursuit, we build Cilantro.

At the core of Cilantro is an online learning mechanism which forms feedback loops with the jobs to estimate the resource to performance mappings and load shifts. This relieves users from the onerous task of job profiling and collects reliable real-time feedback. This is then used to achieve a variety of user-specified scheduling objectives. Cilantro handles the uncertainty in the learned models by adapting the underlying policy to work with confidence bounds. We demonstrate this in two settings. First, in a multi-tenant 1000 CPU cluster with 20 independent jobs, three of Cilantro's policies outperform 9 other baselines on three different performance-aware scheduling objectives, improving user utilities by up to $1.2 - 3.7\times$ and performs comparably to oracular policies. Second, in a microservices setting, where 160 CPUs must be distributed between 19 inter-dependent microservices, Cilantro outperforms 3 other baselines, reducing the end-to-end P99 latency to $\times 0.57$ the next best baseline.

3.1 Introduction

The goal of cluster resource managers is to allocate a finite amount of scarce resources to competing jobs. When doing so, we should ensure that the allocations fulfill the users' and the organization's overall goals. Traditionally, resource allocation policies have aimed to provide fairness [50, 39], maximize resource utilization [189], maximize the amount of work done [50], or minimize queue lengths [201, 129]. However, these policies miss, or at best are imperfect proxies for what

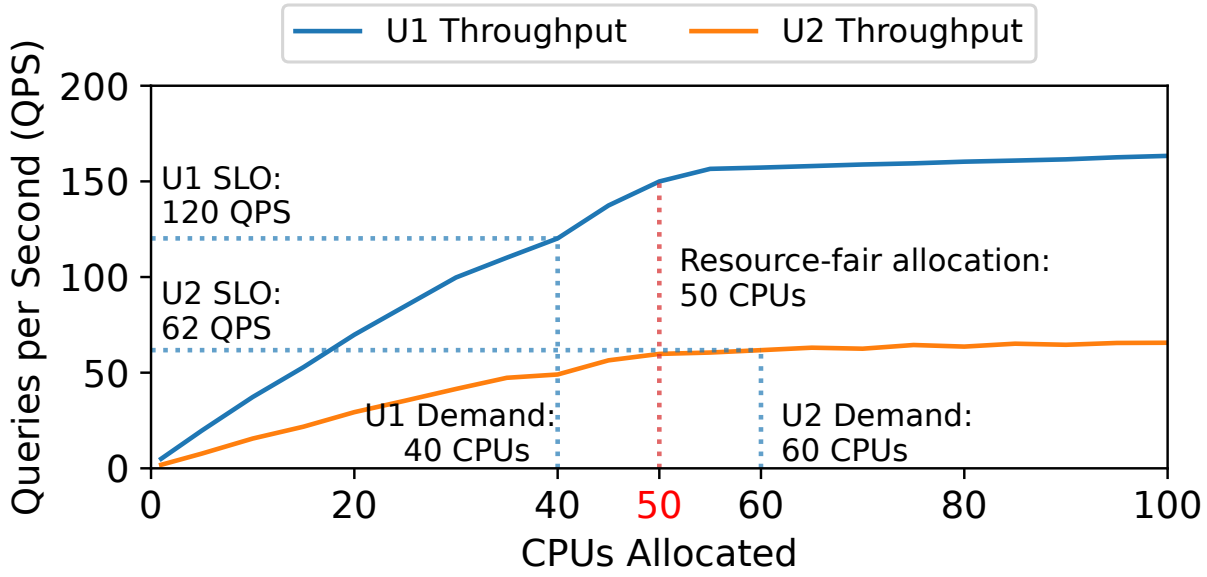


Figure 3.1: Two users, U1 and U2, serving TPC-DS benchmark queries with different resource-throughput mappings and performance goals (SLO). A user's demand is the amount of CPUs needed for her SLO.

matters most to the users: the performance of their jobs in terms of real-world metrics that impact business (e.g. P99 latency or throughput for a serving job). Barring some recent exceptions [25, 199, 44, 80], resource allocation systems have traditionally focused on the resources requested by a job rather than the job's real-world performance from using those resources (henceforth, simply *performance*).

To illustrate the pitfalls of performance-oblivious scheduling, consider an example where two users, U1 and U2, are sharing a cluster of 100 CPUs. They are each serving different sets of TPC-DS[122] queries and care about their throughput: U1's service level objective (SLO) is 120 queries per second (QPS), while the U2's SLO is 62 QPS. If the goal is to satisfy all user's SLOs, how should CPUs be allocated? If it were known that the resource-to-throughput curves of the two users' jobs were as shown in Figure 3.1, a scheduler can allocate 40 CPUs to the first job and 60 to the second. However, in practice, this mapping is usually not available and performance-oblivious scheduler will likely be suboptimal. For instance, a CPU-based fair allocation algorithm would allocate 50 CPUs to each user, which would result in U2 getting just 59 QPS, thus missing its SLO.

Despite extensive theoretical work [39, 50, 82, 83, 61], performance-aware scheduling has remained challenging since the *resource-to-performance mappings* are usually unavailable in practice. To obtain these mappings, past work [180, 38, 199] profile their workloads before execution. Such profiling has three limitations. First, offline profiled resource-to-performance mappings may not reliably reflect a job's performance in a production environment, as it may not capture the interference from other jobs [37] and the server's performance variability [45]. Second, jobs' resource requirements change with time due to varying load (e.g., arrival rate of external queries) and profiling typically cannot account for these changes. Third, such profiling is burdensome for

users and expensive for organizations as it requires a large pool of resources to exhaustively profile a wide range of resource allocations. This informs the *first requirement* for this work: obtain the resource-to-performance mappings in the production environment where the job will be run.

Even if the resource-to-performance mappings are known, the choice of scheduling policy depends on the objective of the end-users (e.g. organization, developers). For instance, suppose in Figure 3.1, we wished to maximize the total throughput of the cluster, instead of trying to satisfy each user's SLOs. In this case, we would allocate $\sim$64 CPUs to U1 and $\sim$36 to U2 for a total throughput of $\sim$212 QPS. As more realistic examples, in multi-tenant clusters, we may wish to use policies which balance between performance and fairness [83, 39, 50]. In contrast, when we provision resources to different microservices of the same application, we are more interested in some end-to-end performance objective, such as application latency, and may wish to allocate more resources to critical microservices which bottleneck performance. These objectives can vary from organization to organization and optimizing for such different objectives requires different allocation policies. However, while end users may find it relatively easy to state their objective (e.g., satisfy all SLOs, maximize throughput), it is harder to design a policy to achieve it. This informs our *second requirement*: support a diverse set of user-defined scheduling objectives.

To address these requirements, we introduce Cilantro, a framework for performance-aware allocation of a single fungible resource type (e.g. CPUs, containers) among competing jobs (Figure 3.2). In Cilantro, end users first declare their desired scheduling objective. To satisfy the first requirement, a pool of performance learners and load forecasters analyzes live feedback from jobs and learns models to estimate resource-performance curves and load shifts for each job. To satisfy the second requirement, Cilantro's scheduling policies, which are automatically derived based on the users' objectives, leverage these estimated models to compute allocations for each job. As the learned models become accurate over time, Cilantro is able to eventually achieve the users' objectives. This obviates the need for an offline model to estimate the required resource allocation for a given performance target, and allows Cilantro to optimize for custom objectives, such as various fairness or performance criteria. This is a marked departure from performance-oblivious policies, those based on unreliable proxy metrics such as CPU utilization and queue lengths, and other heuristic-based policies (using either surrogates [152] or performance metrics [25, 44]) which are designed for very specific scheduling objectives. Cilantro seamlessly enables the implementation of performance-aware policies in two settings: *(i)* multi-tenant resource allocation for independent jobs, and *(ii)* resource allocation for inter-dependent jobs (microservices) within an application.

Our proposed solution solves two key challenges. First, estimating resource-to-performance mappings online can be notoriously difficult due to highly stochastic nature of real-time production environments, unexpected load shifts, especially in the early stages when there is insufficient data. To operate without accurate estimates, Cilantro informs scheduling policies with confidence intervals of its estimates. Policies are designed to account for this uncertainty when making allocation decisions until the estimates become more accurate. Accounting for this uncertainty helps Cilantro conservatively explore the space of allocations making it robust to environment stochasticity and also to the idiosyncrasies specific to the performance models used.

Second, supporting a diversity of objectives in the same framework is challenging. The monolithic design of end-to-end feedback-driven approaches[199, 79, 137] restricts them only the objec-

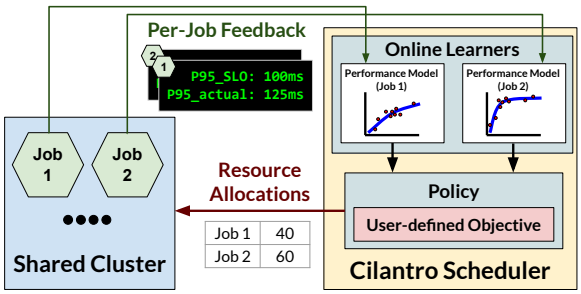


Figure 3.2: Cilantro overview. Cilantro uses continuous feedback to dynamically learn each job's resource-to-performance mappings. An uncertainty-aware resource allocation policy, instantiated for the user's objective, uses these mappings to determine allocations.

tive they were originally designed for. Instead, Cilantro achieves generality in supporting custom objectives by decoupling the learning mechanisms from the allocation policy. This decoupling is necessary as it allows us to account for the effect of each job's performance and load shifts on the objective individually. Moreover, this decoupling has other intangible benefits: it leads to a more transparent design which is easy to debug than monolithic systems which directly optimize for end-to-end performance, and if online job feedback cannot be obtained for a particular job, it is easy to swap the learners with profiled information or other sensible defaults.

We have implemented Cilantro as an open-source extension to the Kubernetes core scheduler, available at `https://github.com/romilbhardwaj/cilantro`. To evaluate Cilantro, first we deploy it on a 1000-CPU multi-tenant cluster which includes a diversity of real-world, latency and throughput-sensitive jobs. On three different allocation objectives, Cilantro's policies are able to outperform 9 other baselines, and is able to compete with oracular policies which know the resource-to-performance mappings a priori on resource efficiency and fairness. When compared to resource-fair allocation, it is able to increase the performance of $1/3$ of users in the clusters by $1.2 - 3.7\times$. Second, we evaluate Cilantro on a 160 CPU cluster where we wish to allocate CPUs to constituent microservices of an application. Here, Cilantro is able to minimize the end-to-end P99 latency of the application to $\times 0.18$ the latency of a resource-fair scheduler and to $\times 0.57$ of the next-best performance-aware baseline.

3.2 Background & Related Work

In this section, we compare Cilantro with prior work. Table 3.1 summarizes the key differences of Cilantro against other resource allocation systems and methods.

Performance oblivious methods: The simplest, yet popular approach to allocating finite resources among competing jobs, is to adopt a *resource fair* policy, which simply divides the resource equally (or proportional to weights) [72, 69, 62, 116]. As this does not account for jobs' resource requirements, it is inadequate in all but the most trivial settings.

Several scheduling frameworks, such as Kubernetes [22], Mesos [64] and YARN [179], relies on users to submit their own resource demand. To execute resource allocations from policies,

	Cilantro	PARTIES[25]	Henge[80]	Autopilot[152]	Jockey[44]	Paragon[37]	Morpheus[76]	DS2[79]	Quasar[38]	FIRM[137]	Sinan[199]	YARN[1
Performance awareness	RW	RW	RW	RW	RW	RW	RW	RW	PM	PM	PM	PO
Works without apriori performance model?	Y	Y	Y	Y	N	N	N	N	Y	Y	N	NA
Supports multiple allocation objectives?	Y	N	N	N	N	N	N	N	N	N	N	Y[1]
Cluster size	Fix	Var	Fix	Var	Fix	Fix	Fix	Var	Fix	Var	Var	Fix

abbr. RW = Real-world metrics, e.g., latency, PM = Proxy metrics e.g., CPU util., PO = Performance oblivious, Fix = Fixed size, Variable size

[1] Supports multiple objectives, but only performance oblivious ones

Table 3.1: Cilantro and related work. Cilantro uses real-world metrics (e.g., latency) to build performance models online, w can be used to derive custom policies for different objectives.

Kubernetes and YARN use resource reservations while Mesos negotiates through resource offers. This requires users to estimate their jobs' resource needs, which can be difficult. They focus on one-way resource allocations and do not provide any mechanisms for the policy to get feedback on application performance. However, recognizing that end users may have varied scheduling objectives, these frameworks support and implement multiple policies.

Methods based on proxy metrics: The most common approach to account for resource requirements relies on proxy metrics (e.g. CPU utilization, work-queue lengths). Quasar [38] offline profiles jobs' proxy metrics, and has a fixed operator-centric policy to maximize cluster utilization. Paragon [37] accounts for resource heterogeneity and inter-job interference to achieve performance guarantees. AGILE[125] models the resource pressure, and uses demand prediction to minimize SLO violations. The above works do not directly account for users' performance goals and optimize for singular objectives.

Methods which use offline profiling: Some work has explored directly incorporating job performance via profiled historical data. Morpheus [76] aims to mitigate performance unpredictability by defining SLOs and satisfying their resource demands by using models based on historical data. Ernest [180] provides methods for estimating performance curves using limited amount of data, but does not study using these estimates for resource allocation under scarcity. Sinan [199] partly uses profiled information for auto-scaling in a cloud environment. Quincy's [69] min-cost flow formulation aims at providing fairness, but relies on offline estimates of data movement costs. For reasons explained in §4.1, offline profiling can be problematic and it is desirable to rely on real-time feedback to determine resource allocations.

Methods which use online feedback: Among related work, some feedback-driven systems account for performance metrics and SLOs in resource allocation. Jockey [44] focuses on meeting latency SLOs for a single job by modeling internal job dependencies to dynamically re-provision resources. Henge [80] defines new utility functions for stream processing workloads and aims to maximize a singular objective – the sum of utility of all jobs. [131] uses application hints in for

prefetching disk blocks in the OS kernel. Gavel [123] is a scheduler for ML training workloads in heterogeneous environments with varying objectives. Since Gavel is focused on ML training, it's policies are designed for throughput and a greedy optimizer computes the optimal allocation for each round. On the other hand, Cilantro supports any metric specified by the user and employs online learning to eventually converge on the optimal allocation. Finally, in a video streaming application, Minerva [124] studies methods for resource allocation so that all end users have the same quality of service. The highly customized policies used in the above works, while adequate to the allocation objectives set out by the authors, are not applicable for diverse cluster objectives which is our goal here.

Variable resource amounts: In other related work, PARTIES [25] allocates resources to jobs within the same server while always satisfying SLOs. If the SLOs of all jobs cannot be met, it evicts one of them to a different server; thus, it does not apply to our setting where there is a fixed amount of resources and eviction is not possible. Indeed, in §3.7 we show that a straightforward adaptation of PARTIES does not work as well. Sinan [199], DS2 [79], Autopilot [152] and FIRM [137] consider performance-aware resource allocation using online feedback when there is elasticity in resource availability, e.g. the cloud. Because these works can scale up to more resources than originally provisioned, they are not directly comparable to Cilantro which operates in a fixed cluster setting. While the cloud is an emerging use case, traditional fixed resource cluster management remains pertinent for privacy and cost reasons. Moreover, the above work focus on specific goals and are not designed to handle general allocation objectives. As an example, FIRM [137] focuses on autoscaling resources for single applications deployed as microservices to minimize end-to-end SLO violations, Cilantro operates differently, reallocating a fixed number of resources according to user-specified objectives, which can include fairness considerations. Additionally, FIRM uses Reinforcement Learning with anomaly injection, in contrast to Cilantro, which focuses on resource-allocation under uncertainty and is agnostic to the learning method used.

3.3 Cilantro Architecture

Cilantro is a performance-aware scheduling framework that can optimize for various scheduling objectives without requiring any a priori knowledge of the resource-performance mapping of the workloads. The design of Cilantro is informed by the following two key insights.

[I1] Offline profiling of resource-performance is insufficient. Performance-aware policies rely on accurate estimates of resource-to-performance mappings and load shifts. Offline profiling of these resource-performance mappings can be inaccurate due to unpredictability in server and application performance [45] and changing traffic patterns [147]. Adapting to these changes necessitates continuously learning and predicting these unknowns in an online manner.

[I2] Decoupling learning mechanisms and policies enables diverse scheduling objectives. As different scheduling policies optimize different criteria, it may be challenging for a scheduling framework to generally support different policy types. Prior work on feedback-driven resource allocation [199, 79, 137] uses an end-to-end model for allocating resources for a fixed objective, such as total utility or cost. Optimizing for a different objective in these systems may require a

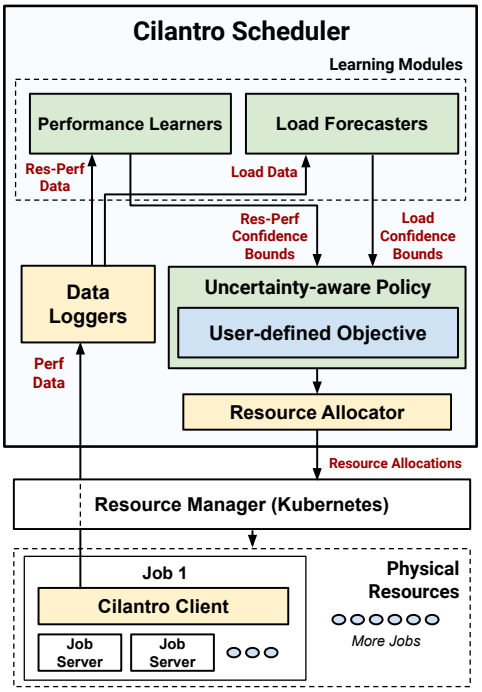


Figure 3.3: The Cilantro scheduler and client architecture. The scheduler generates resource allocations for jobs and the clients collect performance feedback to report to the scheduler.

complete redesign of the system and policy, or at the very least an expensive retraining of their models. Decoupling learning mechanisms from policies allows the model to be learned once and applied to multiple allocation objectives. This decoupling also increases transparency in the allocation decisions made by the scheduler and facilitates debugging.

We leverage these learnings to build Cilantro (Figure 3.3). Cilantro is composed of two key components: the centralized Cilantro scheduler, which is responsible for generating resource allocations, and the Cilantro clients—lightweight sidecars co-located with each job—which fetch a job's performance metrics and send them to the Cilantro scheduler. Informed by **[I1]**, the Cilantro scheduler employs online learning to create increasingly accurate models of job performance and load. Guided by **[I2]**, the policy optimizes a user-defined objective by polling these models for a resource-performance estimates to produce a resource allocation.

Assumptions & terminology. In this work, we will focus on jobs which can scale elastically with the number of resources with corresponding gains in performance. Examples of such workloads include stateless or stateful distributed services (e.g., prediction serving [31], memcached [47], Cassandra [89]), distributed computation (ML training, MPI jobs) and distributed frameworks (e.g. Hadoop [159], Spark [195]). Some of these can be viewed as a collection of several tasks whose job size may vary with time, such as in serving jobs. Each task may refer to a query whose arrival rate may change with time. For jobs with such varying query rates, we will refer to the instantaneous rate of external query arrival as the *load* (measured in queries per second (QPS)). Finally, we assume there is a *fixed amount* of a *single*, *fungible* resource type that must be

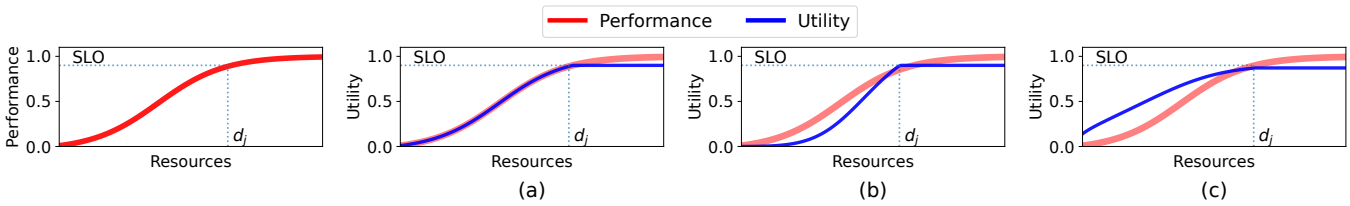


Figure 3.4: Three candidates for SLO-based utility functions. The left-most figure shows a job's performance p_j as a function of the resources (for fixed load). In (a), the utility scales linearly with performance until the SLO, i.e $u'_j(p) \propto \min(p, \text{SLO})$, whereas in (b) it scales quadratically $u'_j(p) \propto \min(p, \text{SLO})^2$, and in (c) it scales with the square-root $u'_j(p) \propto \min(p, \text{SLO})^{1/2}$. Here, (b) captures settings where even small SLO violations are critical while (c) captures settings where small SLO violations are not very significant.

allocated.

Cilantro scheduler: The Cilantro scheduler is designed as a centralized asynchronous event driven system. Event sources include timers, performance updates received from the Cilantro clients, and cluster state updates from the underlying resource manager. Below, we describe the scheduler's modules. Specific implementation details are available in §3.6.

1. Data loggers. Application metrics pushed from Cilantro clients are stored in memory-backed tables. They relay these metrics to the performance learners and load forecasters.

2. Performance learner. The performance learner learns a job's performance as a function of the resource allocation and the load using an associated model. It periodically polls the data logger for new data and updates the model. The learner's update frequency is constrained only by the velocity at which the model can be updated. One instance of a performance learner is maintained per application. A performance learner provides `get-perf-ucb` and `get-perf-lcb` interfaces for a policy, which return upper and lower confidence bounds for the performance as a function of the resources and load.

3. Load forecasters. In many real-world deployments, the job size could vary with time depending on the real-time traffic, which should be accounted for when allocating resources. The goal of the load forecaster is to estimate this load for the duration of a future allocation based on past observed loads via an associated time series model. It offers `get-load-ucb` interface for a policy which returns an upper confidence bound for the future load. Load forecasters are periodically updated by polling from the data loggers.

4. Uncertainty-aware Policy. Policies compute allocations in order to optimize for a user-specified scheduling objective. In an online setting, using direct estimates of the performance may fail as it does not reflect the uncertainty in the model. Therefore, Cilantro's policies leverage confidence intervals of these estimates to account for this uncertainty in a principled manner when making allocation decisions (§3.4).

5. Resource allocator. The resource allocator is responsible for executing the resource allocations by interfacing with the underlying cluster manager. This module is driven via an allocation expiry event, upon which it invokes the policy's `compute-alloc` method and allocates the resources. Allocation expiry events are raised based on a timeout, resulting in a new round of

allocations. In practice, the duration of an allocation round is limited by the agility of the environment. Since scaling jobs requires time, changing resource allocations too frequently can result in job thrashing (having to scale down before it has a chance to utilize new resources).

Cilantro client: The Cilantro client is a lightweight side-car container whose purpose is to to poll the job to get its current performance, process it, and publish it to the scheduler's data loggers. The primary task for the client is to extract metrics from their assigned job. Many systems expose REST endpoints to query system performance [87, 143], but often the applications also use monitoring tools such as Prometheus or Grafana. Depending on the job, the performance metric extraction logic is specified by the users. In §3.5, we describe built-in fallback options if job metrics are not available.

3.4 Policies

We now describe our policies for performance-aware resource allocation in two settings: multi-tenant resource allocation in a fixed cluster (§3.4), and allocating finite resources to constituent microservices of an application (§3.4).

Set up & notation: We will denote the number of jobs (or microservices) by n, the amount of resources by R, and an allocation by $a = (a_1, \ldots, a_n)$, where a_j is the amount of resources allocated to job (or microservice) j. A scheduler should allocate these resources so that $\sum_{j=1}^{n} a_j \leq R$.

Resource allocation in shared clusters

Cilantro supports two classes of performance-aware allocation objectives in the multi-tenant setting: welfare-based, and demand-based. Our primary contributions are in §3.4 where we derive uncertainty-aware online variants of these policy classes. But first, we will review some common examples of such objectives in §3.4. For what follows, we will need to define the *performance*, *demand*, and *utility* of a job.

Performance: The *resource/load-to-performance mapping* (henceforth simply performance or performance mapping) p_j of a user's job j refers to some raw metric of interest, which, say, can be obtained from a monitoring tool. We write the performance $p_j(a_j, \ell_j)$ as a function of the resources received a_j and the load ℓ_j. As we are allocating a single resource type, a_j is a single number, as is ℓ_j. For example, in a serving job with a P95, 100 ms latency SLO, the performance may be the fraction of queries completed in under 100 ms, and the load may refer to the external arrival rate of queries.

Demand: If a job has a well-defined SLO, we define the *demand* d_j to be the minimum amount of resources needed to achieve this SLO. The demand depends on the job's performance curve p_j, SLO, and load ℓ_j.

Utility: The *utility* u_j of a job is the *practical value* derived due to its performance. Generally, u_j is a non-decreasing function of the performance and we can write $u_j(a_j, \ell_j) = u'_i(p_j(a_j, \ell_j))$ for some non-decreasing function u'_j.

Examples of utilities. The simplest option is to set the utility to be equal to the performance $u_j = p_j$, i.e., u'_j is the identity. However, we may also choose a utility which is more applicable when there are well-defined SLOs. Fig. 3.4 illustrates three candidates for u'_j: the maximum utility for any job is set to 1, which is achieved for any performance greater than the SLO; for performances below the SLO, we may set the utility to *(a)* decrease proportionally with SLO violation, *(b)* decrease sharply in settings where small SLO violations are critical (e.g., with external customers where SLO violations can lead to penalties [6] and a loss of credibility), *(c)* decrease gradually when small SLO violations are not critical (e.g., soft SLOs internal to an organization). Such utility forms which are 'clipped' at the SLO provide a simple way to compare jobs with heterogeneous performance metrics and SLOs, such as latency and throughput. Prior work have also used similar forms of utility [80, 51, 184]. For these reasons, our experiments also use these forms, although we emphasize that Cilantro can handle any utility form which increases with performance.

Review of multi-tenant allocation when performance mappings are known

We will first review two classes of multi-tenant allocation objectives supported in Cilantro—welfare-based and demand-based—and three examples of such objectives. In §3.4, we will develop online learning policies that achieve the same objectives when performance mappings are unknown.

Welfare-based objectives: These policies aim to maximize a given cluster-wide *welfare* function W, which is a function of the utility of each job, i.e., $W = W(u_1, \ldots, u_n)$. Below, we describe two common welfare-based objectives.

(i) Social welfare (a.k.a. Kelly mechanism [83]): We choose the allocation a which maximizes the social welfare (the average utility), i.e. $a = \text{argmax}\, W_{\text{S}}$, where,

$$W_{\text{S}} = \frac{1}{n}\sum_{j=1}^{n} u_j(a_j, \ell_j) = \frac{1}{n}\sum_{j=1}^{n} u'_j(p_j(a_j, \ell_j)). \tag{3.1}$$

As we show in Figure 3.5, this notion of fairness allocates more resources to "high-performing" users, i.e those who can generate large utility with a small amount of resources.

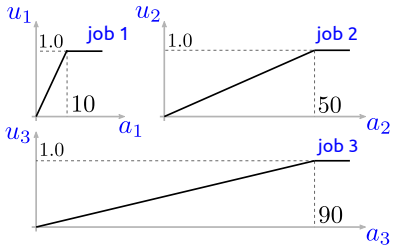
(ii) Egalitarian welfare: Here, we choose the allocation a which maximizes the egalitarian welfare (minimum of all utilities), i.e. $a = \text{argmax}\, W_{\text{E}}$, where

$$W_{\text{E}} = \min_{j\in\{1,\ldots,n\}} u_j(a_j, \ell_j) = \min_{j\in\{1,\ldots,n\}} u'_j(p_j(a_j, \ell_j)). \tag{3.2}$$

This allocates more resources to "struggling" jobs which need more resources to achieve large utility (Figure 3.5).

Demand-based policies: These policies apply when jobs have a well-defined SLO and it is possible to define its demand d_j. Such policies will compute allocations based on the demands of all jobs. This requires knowledge of the demand, which in turn depends on the performance mapping.

(iii) No justified complaints (NJC) fairness [39, 40, 61]: One class of demand-based policies which adopt the NJC fairness paradigm guarantee an equal share of R/n for each job. If the job's demand is larger than R/n, it is allocated at least (but possibly more than) this share. But, if the



Policy	allocation (utility) job 1		job 2		job 3		metrics W_{S}	W_{E}	F_{NJC}
Social welfare	10	(1.0)	50	(1.0)	0	(0.0)	0.67	0.0	0.0
Egalitarian welfare	4	(0.4)	20	(0.4)	36	(0.4)	0.4	0.4	0.4
NJC fair	10	(1.0)	25	(0.5)	25	(0.28)	0.59	0.28	1.0
Resource fair	20	(1.0)	20	(0.4)	20	(0.22)	0.54	0.22	1.0

Figure 3.5: Comparison of the three (oracular) fair allocation criteria described in §3.4 in a synthetic example with 60 CPUs. *Left:* Utility curves for three jobs. The y axis is the utility and the x-axis is the number of resources. For simplicity, we have ignored the loads and assumed that utilities increase linearly up to the demand. The total demand is 150, whereas only 60 resources are available. *Right:* The allocations and utilities for each job under the three criteria. We have also shown the W_{S} (3.1), W_{E} (3.2), and F_{NJC} (3.3) metrics for each policy.

job's demand is smaller, the excess resources may be allocated to other jobs to improve overall resource usage. A user can have no justified complaints since they are either guaranteed to satisfy their SLOs or their utility will be larger than if they were to have R/n resources. To quantify this, we define the following metric. The term inside the minimum measures the utility achieved by job j with allocation a_j relative to the utility when using its fair share of R/n resources.

$$F_{\text{NJC}} = \min_{j \in \{1,\ldots,n\}} \frac{u_j(a_j, \ell_j)}{u_j(R/n, \ell_j)} = \min_j \frac{u'_j(p_j(a_j, \ell_j))}{u'_j(p_j(R/n, \ell_j))} \tag{3.3}$$

In contrast to metrics such as the Jain's index[71], F_{NJC} accounts for users' performance when evaluating fairness. This metric has a maximum value of 1. Below, we describe a demand-based policy [39] which achieves $F_{\text{NJC}} = 1$ while also using the resources efficiently as also shown in Figure 3.5.

An NJC policy: This policy proceeds iteratively. In the first round, it sets each user's "share" to be R/n. It allocates d_j to each user j for whom d_j is smaller than the share. If n' users were allocated R' resources in the first round, in the second round it sets each user's share to be $(R-R')/(n-n')$. It repeats this until all the remaining users' demands are larger than their share. It then divides up the remaining resources equally among the remaining users. While this policy may not maximize any welfare, it achieves Pareto-efficient user utilities. Another advantage of this policy is that it is strategy-proof, i.e a user does not gain additional utility by falsely stating their demand [50, 61, 81].

This concludes our review of multi-tenant resource allocation objectives when performance mappings are known. We mention that prior work have used these objectives in various contexts with custom utilities. For instance, social welfare has been used in stream processing [80] and wireless networks [176], egalitarian welfare in video streaming [124], and several NJC policies are implemented in Mesos [64].

Online learning policies in Cilantro

We will now develop our online policies. Our policies will operate on lower and upper confidence bounds obtained from the load forecasters and performance learners instead of the direct estimates; doing so accounts for the uncertainty in the learned models and encourages a policy to conservatively explore the space of allocations until the estimates become accurate. Cilantro's policies will proceed sequentially in allocation rounds. On round r, Cilantro chooses an allocation $a^{(r)} = (a_1^{(r)}, \ldots, a_n^{(r)})$ based on the feedback from all jobs up to now and the specific scheduling objective.

Welfare-based online policies: For welfare-based policies, Cilantro adopts the optimism in the face of uncertainty (OFU) principle [20]. OFU stipulates that, to maximize an uncertain function, we should choose actions which maximize an upper confidence bound (UCB) on the function. Both theoretically and empirically, OFU is known to outperform other strategies which use direct estimates or those which are pessimistic (i.e. maximize lower confidence bound). An in-depth exploration of OFU is beyond the scope of this work, but we refer the reader to relevant literature (e.g. [9, 21, 163, 55]).

While OFU is a well established design paradigm, most OFU policies are designed for end-to-end systems which output a single reward signal. Adapting OFU for general welfare-based policies requires studying how the uncertainty in the performance and load translate to a UCB $\widehat{W}$ on the welfare W which we wish to maximize. Since W is non-decreasing in the utilities u_j, we can obtain a UCB for W by plugging in UCBs $\widehat{u}_j$ for the utility u_j, i.e $\widehat{W} = W(\widehat{u}_1, \ldots, \widehat{u}_n)$. Similarly, since u_j is non-decreasing in the performance we can obtain a UCB by plugging in a UCB $\widehat{p}_j$ for p_j, i.e $\widehat{u}_j = u_j'(\widehat{p}_j)$. This leads to the following choice of allocation on round r.

$$a^{(r)} = \underset{a \in \mathcal{A}^{(r)}}{\operatorname{argmax}} W\left(u_1'\big(p_1\big(a_1, \widehat{\ell}_1\big)\big),\ \ldots,\ u_1'\big(p_1\big(a_n, \widehat{\ell}_n\big)\big)\right) \tag{3.4}$$

Above, since the exact load cannot be known, we conservatively over-estimate it via a UCB $\widehat{\ell}_j$ on the load. Here, $\mathcal{A}^{(r)}$ is the allocation space on round r which is defined by two constraints: first, the total allocation cannot be larger than R, i.e. $\sum_j a_j \leq R$; second, the current allocation cannot deviate too much from the previous allocation, i.e. $a_j^{(r-1)} - B \leq a_j \leq a_j^{(r-1)} + B$ for all j, where B is a parameter to be specified. We impose the second constraint since large changes to allocations can have unpredictable effects on a job's performance; moreover, they take a long time to actuate, resulting in unreliable feedback while resources are being scaled up/down.

To optimize (3.4), one can use any off-the-shelf optimizer such as evolutionary algorithms, hill climbing, or integer programming which can handle the linear constraints for $\mathcal{A}^{(r)}$. In our implementation, we used an evolutionary algorithm (details in the appendix). Finally, we describe instantiations of this principle for the two welfare-based policies we saw in §3.4.

(i) Cilantro-SW: To emulate the social welfare policy in §3.4, on round r, we use the UCB for $\hat{l}$ for load and $\hat{p}$ for performance. Thus, we choose an allocation

$$a^{(r)} = \underset{(a_1,\ldots,a_n) \in \mathcal{A}^{(r)}}{\operatorname{argmax}} \sum_{j=1}^{n} u_j'\big(\,\widehat{p}_j(a_j, \widehat{\ell}_j)\,\big).$$

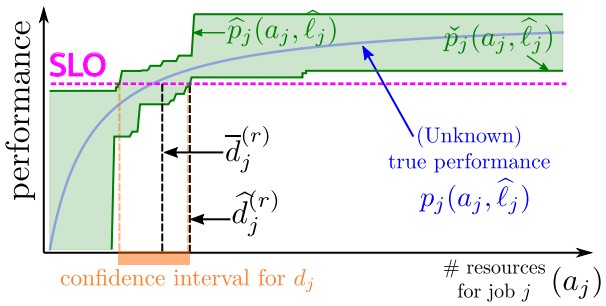


Figure 3.6: Illustration of Cilantro's uncertainty-aware demand-based policies. We first obtain a UCB $\widehat{\ell}_j$ from the load forecaster, which ensures that we have a conservative estimate on the job's load. In the figure, the x axis is the amount of resources a_j that could be allocated to job j. We show the SLO (pink), the slice of the unknown performance curve (blue) when the load is $\widehat{\ell}_j$, and the confidence region obtained from past data (green). The LCB $\check{p}_j$ and UCB $\widehat{p}_j$ on $p_j(a, \widehat{\ell}_j)$ are given by the lower and upper boundaries of the confidence region (solid green lines). A confidence interval for the demand (orange) can be obtained by the region where $\widehat{p}_j, \check{p}_j$ intersect the SLO line. To obtain a recommendation, we compute a UCB $demhatjr$ on the demand (where SLO intersects $\check{p}_j$) and $\overline{d}_j^{(r)}$ via equation (3.5).

(ii) Cilantro-EW: To emulate the egalitarian welfare policy in §3.4, on round r, we choose an allocation

$$a^{(r)} = \underset{(a_1,\ldots,a_n)\in\mathcal{A}^{(r)}}{\operatorname{argmax}} \min_{j\in\{1,\ldots,n\}} u_j'\big(\widehat{p}_j(a_j, \widehat{\ell}_i)\big),$$

Demand-based online policies: For demand-based policies, on round r, we will use the confidence intervals from the performance learners and load forecasters to obtain conservative recommendations $d_j^{(r)}$ for job j's demand. Then, we compute the allocations $a^{(r)}$ for this round by invoking the same demand-based policy with the recommended demands $\{d_1^{(r)}, \ldots, d_n^{(r)}\}$ instead of the true demands.

Our method for obtaining demand recommendations is based on [81]. To describe this in more detail, observe that for demand-based policies it is sufficient to accurately estimate the demand well, i.e. it is not necessary to learn the entire performance mapping well. We have illustrated our strategy for obtaining the demand recommendation in Figure 3.6. First, we will denote by $\widehat{d}_i^{(r)}$, the UCB for the demand obtained as shown in Figure 3.6. As a conservative choice for this demand, we may wish to choose $\widehat{d}_i^{(r)}$ as the recommendation. However, we found that in practice this was overly conservative and the resulting allocations were very slow to adapt to feedback. Therefore, we also wish to use a more aggressive exploration strategy to reduce the uncertainty in our *demand*. We use:

$$\overline{d}_j^{(r)} = \underset{a_j}{\operatorname{argmax}} \min\big(\widehat{p}_j(a_j, \widehat{\ell}_j) - \text{SLO},\ \text{SLO} - \check{p}_j(a_j, \widehat{\ell}_j)\big) \tag{3.5}$$

To illustrate this rule, consider Figure 3.6 where $\min(\widehat{p}_j - \text{SLO},\ \text{SLO} - \check{p}_j)$ is negative for large allocations when the performance LCB $\check{p}_j$ is larger than the SLO and for small allocations where

the performance UCB $\widehat{p}_j$ is smaller than the SLO. By maximizing (3.5), we are choosing points inside the confidence interval for the demand where both $\breve{p}_j, \widehat{p}_j$ are further away from the SLO; so if job j were to receive $\overline{d}_j^{(r)}$ resources, then we are most likely to reduce the demand uncertainty. However, choosing $\overline{d}_j^{(r)}$ as the recommendation can lead to overly aggressive exploration so our final recommendation $d_j^{(r)}$ is then obtained via,

$$d_j^{(r)} = \text{clip}\Big(\beta\widehat{d}_j^{(r)} + (1-\beta)\overline{d}_j^{(r)},\ \ d_j^{(r-1)} - B,\ d_j^{(r-1)} + B\Big) \tag{3.6}$$

Here, $\beta \in (0,1)$ is a parameter to trade-off between $\widehat{d}_i^{(r)}$ and $\overline{d}_i^{(r)}$. We clip this value between $d_j^{(r-1)} - B$ and $d_j^{(r-1)} + B$ to control wide deviations in resource allocations (similar to before). Next, we formally state Cilantro's instantiation of the demand-based NJC procedure described in §3.4.

(iii) Cilantro-NJC: Here, we simply compute the recommended demand via (3.5), and then invoke the NJC procedure described in §3.4, In §3.7 we show that Cilantro-NJC retains some of the strategy-proofness properties of NJC.

Microservice resource allocation

Now, we will look another use-case for Cilantro, where we wish to optimize an end-to-end performance metric p of an application composed of several interdependent microservices (jobs). Examples for p include the total throughput of the application, the negative P99 latency, or even any combination of the two. Here, the entire fixed set of resources is available to the application and must be allocated between microservices for to maximize p. There are two main differences in this setting when compared to the multi-tenant setting which introduces new challenges. First, while assuming jobs run by different users are independent is reasonable when we aim to optimize for fairness, this is no longer true now since microservices within an application may have complex dependency graphs (see Figure 3.12-Left). Second, while an application's performance is clearly tied to the performance of individual microservices, it is not possible to write it explicitly, as we did for the social or egalitarian welfare.

We overcome these challenges by modeling the end-to-end performance p as a direct function of the allocation to each microservice and the external load faced by the application. That is, we write $p(a, \ell)$, where, $a = (a_1, \ldots, a_n)$ is a vector of allocations for each microservice and ℓ is the external load on the application. On allocation round r, our online learning policy, which adopts the OFU principle, chooses an allocation vector which maximizes an upper confidence bound $\widehat{p}$ on the performance obtained from the performance learners:

$$a^{(r)} = \underset{a \in \mathcal{A}^{(r)}}{\text{argmax}}\, \widehat{p}(a, \ell). \tag{3.7}$$

While this circumvents accounting for individual microservice performance and dependency graphs, we now face the challenge of optimizing for an n–dimensional allocation with just one feedback signal. In contrast, in the multi-tenant setting we had more feedback (one for each job).

3.5 Discussion

We now present a discussion on Cilantro's operation under various adversarial conditions that may occur in deployment.

When online job feedback is unavailable. Cilantro provides three fallback options when online feedback is not available. First, Cilantro allows a user to use a profiled model (using historical data) instead of online feedback. Second, it allows using proxy metrics from the Kubernetes API instead of real-world performance. In such cases, a user should specify how these proxies are tied to their utility and/or demand. Third, if neither of these is possible, we allow the user to directly submit an estimate for their resource demand which will then be fed to the policy when determining allocations. In such cases, we assume that utility increases linearly up to the demand when computing allocations. We evaluate this fallback option in §3.7. Due to Cilantro's decoupled design, these fallback options can be effected with simple modifications to a job's performance learner.

Learning in unpredictable environments. Some situations, such as unexpected load spikes for web services or interference between jobs, are fundamentally hard to predict. Cilantro's uncertainty-aware design provides a degree of resilience against these unpredictable changes, as we show in it's robustness to noise in load and resource demand estimates in Section 3.7. However, continued extreme fluctuations in the loads can negatively impact Cilantro's performance. To avoid hysteresis when reallocating resources, future work can explore averaging loads over dynamically sized windows or including rules to temporarily override Cilantro's policy.

Limitations and Future Work. Cilantro currently supports allocating only a single resource type. In our current implementation, multiple resource types can be bundled into grouping units, such as VM SKUs with a fixed ratio of CPU, Memory and GPUs, which can then be scheduled by Cilantro. However, such bundling is not always possible, especially when different jobs have different resource requirements. Extending Cilantro to handle multiple resource types is possible for welfare-based policies. However, learning and optimization can be challenging since the search space is now very large. Another related limitation is that Cilantro cannot handle non-fungible resource types. Cilantro also does not support online learning versions of market-based resource allocation policies in the multi-tenant setting [196, 88, 178]. These are avenues for future work to improve Cilantro. Cilantro also assumes utilities increase with increasing resources, however some workloads may demonstrate inverse scaling, especially when allocated resources become fragmented across physical nodes. Future work can relax this assumption by applying learning techniques robust to non-convex utility shapes. We also note that Cilantro can support multiple SLO parameters (e.g., for an inference job, ensuring a minimum latency and accuracy) by wrapping them in a single utility function, and the design of such utility functions can be explored by future work.

3.6 Implementation

The Cilantro scheduler is implemented in 7600 lines of Python code, as a standalone scheduler for Kubernetes. Resource reallocation events are triggered by a timer-based event, which is raised every 2 minutes in our experiments. This window was chosen based on the fact that Kubernetes pods could be created and destroyed in 5-15 seconds. A 2 minute allocation round is long enough for the pod to reach its steady state that performance metrics from the job would be reliable, while at the same time frequent enough to adapt to changes in the load and learned performances.

To execute updated resource allocations received from policies, we horizontally scale the workloads by adding more replicas to their Kubernetes deployment. Newly created pods rely on the Kubernetes service discovery mechanism to connect to the workload's other servers. The workload is responsible for load balancing queries onto the new servers. Workloads write logs to a volume shared with the sidecar cilantro client. The client parses performance metrics and then publishes them to the scheduler over gRPC. These messages also act as heartbeats to inform liveness to the scheduler.

The frequency of performance feedback depends on the application and the environment. For instance, database serving jobs may report feedback multiple times in a minute, while ML training jobs may do so once every few minutes. To avoid bottlenecks, we use an asynchronous design for Cilantro where each component operates in a push or pull based framework. This allows high frequency components to operate at their maximum rate while allowing slower components, such as learners for low-frequency jobs or cluster managers, to be polled when required.

Specifying utilities and objectives. Utilities of jobs are calculated based on the performance metrics collected by the Cilantro clients in the last resource allocation round. To compute the utilities, application developers specify utility as a python method which operates on a list of floating point numbers representing the performance metrics observed in the previous resource allocation round. Similarly, the scheduling objective (e.g., social welfare from §3.4) is also defined by the cluster operator as a python method operating on the list of utilities from all jobs.

Learning models and load forecasters. For the multi-tenant setting, we used a tree-based binning estimator [81, 21, 58] with Lipschitz constant 10 for each job's resource-to-performance estimation. This is a simple and computationally efficient estimator, but does not work well in high dimensions. Therefore, for the microservices setting where we have a high dimensional estimation challenge, we use kernel ridge regression [200, 183] with a Matern kernel with smoothness parameter set to 2.5. In both settings, for the load forecasters, we use an autoregressive moving average (ARMA) model [108] with autoregressive order 1 and moving average order 1. Finally, all confidence bounds were computed at the 90% level, meaning that the probability that the true parameter lies between the upper and lower confidence bounds is 90%. We used the above learning models since they are simple and have few tunable hyperparameters. With Cilantro's modular design, these can be easily swapped with any other model as long as they provide reliable uncertainty estimates.

Other policy parameters: For all our policies, we set the parameter B which controls the deviation from the previous allocation to 10. For demand-based policies, we set the parameter β which trades off between conservative and aggressive exploration to $3/4$. For the welfare-based

policies in §3.4 and the microservices use case in §3.4, we use evolutionary algorithms to optimize the UCBs. The exact implementation is described in the appendix.

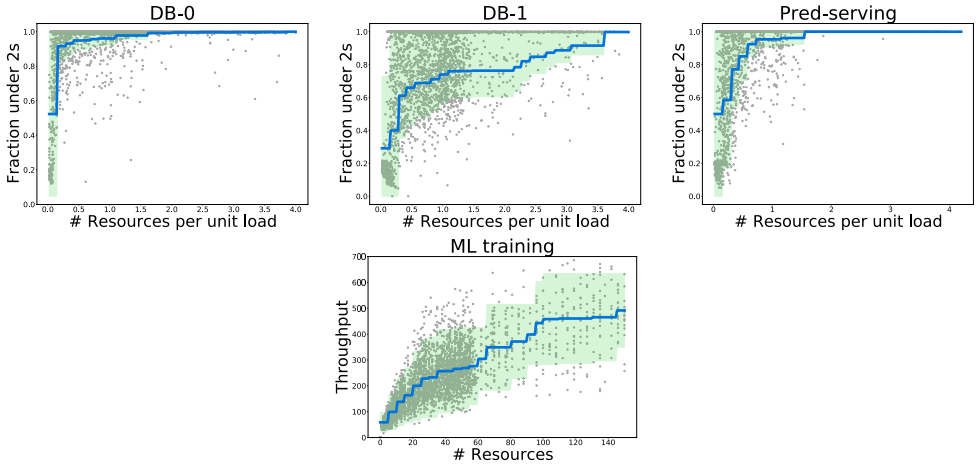


Figure 3.7: Performance vs resource-allocation-per-unit-load obtained after profiling the database querying, predicition serving and ML training workloads. The blue curve is the average performance value and the shaded region is the 2σ confidence interval. For the latency-based workloads (DB-0, DB-1, and prediction serving), we show the number of resources per unit load (arrival QPS) on the x-axis and the fraction of queries completed under 2s on the y-axis. For the ML training workload, we show the number of resources on the x axis the amount of data processed per second on the y-axis. To obtain accurate estimates, we sampled low resources allocations more densely.

3.7 Evaluation

We evaluate Cilantro in two settings described in §3.4.

1. In the multi-tenant setting, Cilantro's online learning policies, which do not start with any prior data, are competitive with oracular policies which have access to jobs' resource to performance mappings obtained after several hours of profiling. Moreover, they outperform 9 other baselines on the metrics outlined in §3.4.
2. In the microservices setting, Cilantro is able to support the completely different objective of minimizing end-to-end latency. It outperforms three other baselines and reduces the P99 latency to $\times 0.57$ that achieved by the next best performance-aware baseline.
3. In our microbenchmarks, we show that Cilantro's allocation policies are inexpensive, evaluate its fallback options when performance metrics are unavailable, and demonstrate its robustness to errors in feedback and choices for performance learner and forecaster models.

Multi-tenant cluster sharing

We first evaluate Cilantro's multi-tenant policies (§ 3.4) on a 1000 CPU cluster shared by 20 users.

Workloads. We use three classes of workloads—database querying, prediction serving and machine learning training—which are used to create multiple jobs. The database querying workload runs TPC-DS [122] queries on replicated instances of sqlite3 database and uses the query latency as the performance metric. The prediction serving workload runs queries on a ML model (random forest regressor) trained on the news popularity dataset [46]. The ML training workload trains a neural network on the naval propulsion [30] dataset using stochastic gradient descent. The database querying and prediction serving workloads use the query latency as the performance metric while ML training uses batch throughput to measure performance. Resource-performance mappings for informing the oracle baselines in §3.7 were obtained through offline profiling of all workloads. These profiles are visualized in Figure 3.7. More details of the workloads, including workload-specific parameters are available in the appendix.

Traces. Queries to the database and prediction serving workloads are dispatched by a trace-driven workload generator. We use the Twitter API [177] to collect a trace of tweet arrival rates at Twitter's Asia datacenters; to bring to parity with our cluster, we subsample the arrival rate by a factor of 10. For the ML training workload, we draw queries from an essentially infinite pool to create a constant stream of work.

Experimental set up. We use a cluster of 250 AWS m5.xlarge instances (4 vCPUs each). The Cilantro scheduler runs on its own dedicated m5.xlarge instance. We use the above 4 workloads to create 20 jobs as follows: 10 database jobs with P90, P90, P90, P90, P95, P95, P95, P95, P99, P99 latency SLOs of 2s; 3 prediction serving jobs with P90, P90, and P95 latency SLOs of 2s; 7 ML training jobs with throughput SLOs of 400, 400, 450, 450, 500, 500, and 500 QPS. To reflect settings where small SLO violations may be either critical or inconsequential, we discount the utility via one of the three options in Fig. 3.4 for each job. Detailed information on the users' jobs is given in the appendix. The estimated total amount of resources based on the median demand was 1637 CPUs; hence, even at full capacity, not all users can satisfy their SLOs. We evaluate all baselines for 6 hours.

Baselines

Oracular policies. We implement the three policies in §3.4 with oracular access to the true performance mappings (obtained by exhaustively profiling workloads for at least 4 hours). They are Oracle-SW, Oracle-EW, for maximizing social/egalitarian welfare and the Oracle-NJC fairness policy.

Cilantro policies. We evaluate Cilantro-SW, Cilantro-EW, and Cilantro-NJC, as described in Sec. 3.4.

Other heuristics. We implement four methods for fairness and maximizing welfare. While not based directly off specific prior work, such methods are common in the scheduling literature [31, 56]. Resource-Fair simply allocates an equal amount of resources to each job. EvoAlg-SW and EvoAlg-EW are evolutionary algorithms for social and egalitarian welfare; the same procedure used for Cilantro's welfare policies, but now operating directly on the performance metrics. Greedy-EW starts by allocating resources equally; on each round, it evaluates job utilities in the previous

round and takes away one CPU each from the top half of the users who had high utility and allocates it to the bottom half.

Baselines from prior work. We adapt five feedback-driven methods from prior work - Ernest [180], Quasar [38], Minerva [124], Parties [25] and MIAD (Multiplicative-Increase/Additive-Decrease) [27]. In particular, we note that applying the Parties notion of migration in our setting would imply moving the job to a different cluster or increasing the size of the cluster, both of which are beyond scope for this fixed cluster setting. Details on the specific adaptations are available in the appendix.

Results & Discussion

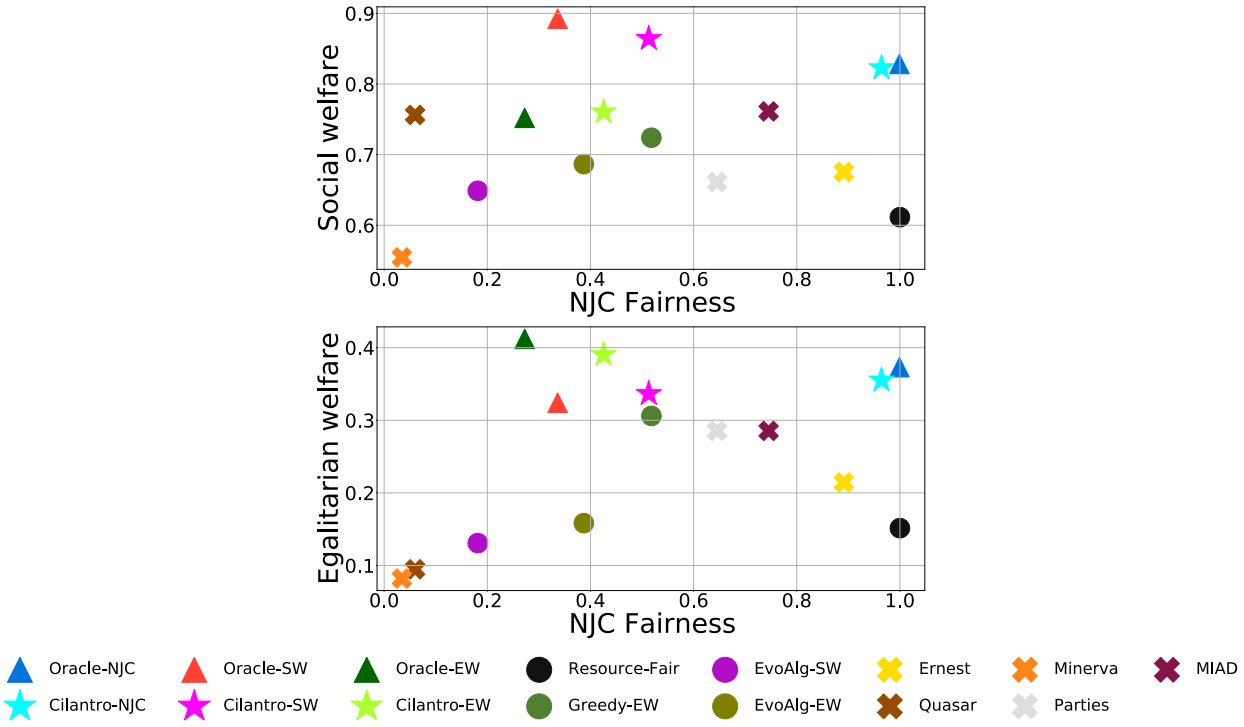


Figure 3.8: NJC fairness vs the social and egalitarian welfare (see §3.4) for all policies. We report the average value over the 6 hour period. Higher is better for all metrics, so closer to the top right corner is desirable. The Oracle-SW, Oracle-EW policies optimize for the social and egalitarian welfare when the performance mappings are known and Oracle-NJC achieves maximum fairness while improving cluster usage. The corresponding Cilantro policies are designed to do the same without a priori knowledge of the performance mappings.

Evaluation on performance-aware fairness metrics. We first compare all 15 baselines on the social welfare (3.1), egalitarian welfare (3.2), and the NJC fairness criteria (3.3). Fig. 3.8 illustrates the results by plotting the time-averaged NJC fairness vs the two welfare criteria. Table 3.2 (in the appendix) tabulates these values explicitly with error bars. While the oracular methods perform

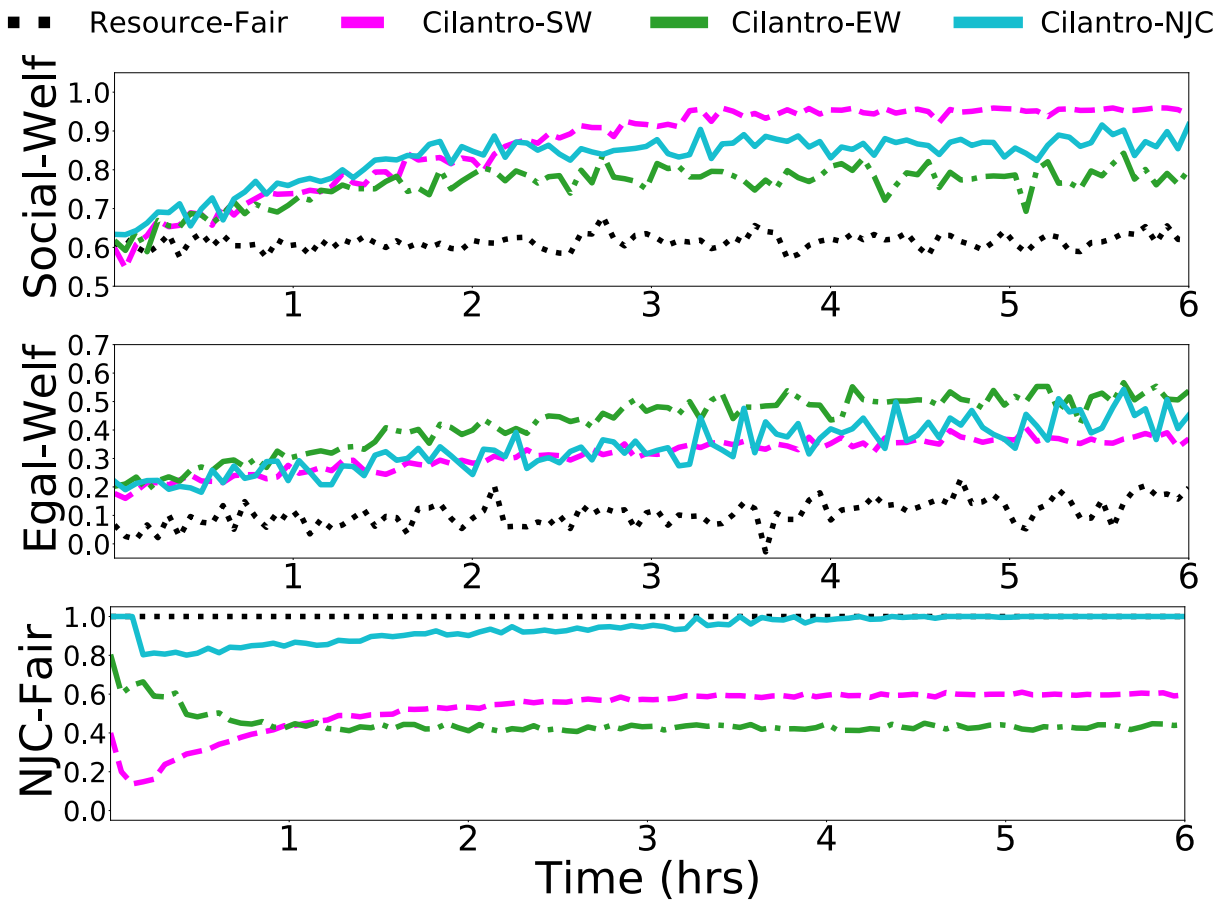


Figure 3.9: Convergence over time of social, egalitarian welfares and NJC fairness for the three Cilantro policies.

best on their respective metrics, we find that the online learning policies in Cilantro come close to matching them. Resource-Fair achieves a perfect NJC score by definition, but performs poorly on social and egalitarian welfare as it is performance oblivious.

We found that Greedy-EW, Parties, and MIAD were sensitive to the amount by which we changed the allocations based on feedback; when tuning them, we found that they were either too slow or too aggressive when responding to load shifts. Next, the learning models used by Quasar and Ernest were not able to accurately estimate the demands in our experiment. Finally, the evolutionary baselines were inefficient, taking a long time to discover the optimal solution. They, however, were effective within Cilantro's welfare policies when you need to optimize a cheap analytically computable function as they can be run for several iterations.

Despite our general approach, Cilantro's policies are able to outperform Minerva and Greedy-EW which are designed specifically to maximize egalitarian welfare. It also outperforms generically designed evolutionary algorithms for the social and egalitarian welfare. While it may indeed be possible to design more efficient fine-tuned policies for a given objective, the flexibility pro-

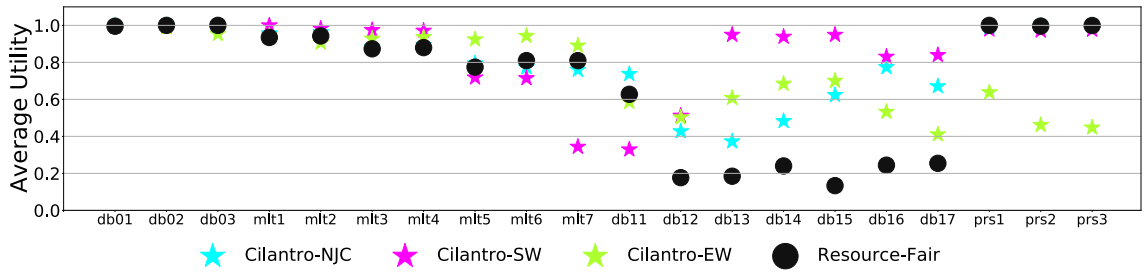


Figure 3.10: The average utility achieved by the 20 jobs for the three online learning methods in Cilantro and Resource-Fair. Here, `db0x`, `mltx`, `db1x`, and `prsx` refers to jobs using the DB-0, ML training, DB-1, and prediction serving workloads from § 3.7.

vided by Cilantro's approach is beneficial to end users. It should not be surprising that Cilantro outperforms other systems such as Ernest, Quasar, Parties, and MIAD as our policies are designed to explicitly optimize for these objectives. *But this is precisely the goal of Cilantro*. End-users can declare their desired objective, and Cilantro will automatically derive policies to achieve them.

To illustrate how Cilantro improves with feedback, in Fig. 3.9, we have shown how the three objectives evolve over time for Cilantro's policies. Resource-Fair trivially achieves $F_{\text{NJC}} = 1$ at start since our initial allocation is always 50 CPUs to each job (i.e Resource-Fair). However, it does poorly on welfare due to poor cluster usage. The goal behind Cilantro-NJC is to achieve $F_{\text{NJC}} = 1$ while also achieving good cluster usage. This causes the initial drop in performance for Cilantro-NJC as it explores better allocations that still maximize F_{NJC}.

Table 3.2 presents the detailed results of our multi-tenant cluster resource sharing evaluation. This table adds a metric which measures the useful resource usage.

$$\text{Useful resource usage} = \sum_{j=1}^{m} \min(a_j, d_j) \tag{3.8}$$

Here, the d_j is user j's resource demand. This demand-based metric, measures how much *useful* work is being done by the cluster as allocations beyond the demand do not increase a user's utility (see Fig. 3.4). We find that Cilantro's policies achieve the maximum useful resource usage in their respective classes. This is because learning resource demands allows Cilantro to reallocate resources from jobs which have already achieved maximum utility to jobs which can benefit from increased resources.

Individual user utilities. To delve deeper into the trade-offs of the three paradigms discussed in §3.4, we have shown the individual user utilities achieved by these three policies in Fig. 3.10. We see that both the social and egalitarian welfare policies result in some users being worse off than receiving their fair allocation of $1000/20 = 50$ CPUs. This results in an NJC fairness violation. In contrast, in Cilantro-NJC, users are at most marginally worse off than their fair share. However, a third of the users achieve a noticeably higher utility than their fair share utility, with more than $3\times$ for a few of them. We also see that Cilantro-EW has maximized egalitarian welfare by taking resources away from those who achieve high utility and giving it to those who do not, while Cilantro-SW has maximized social welfare by allocating more resources to jobs that can quickly achieve high utility.

Policy	Social Welfare, (W_{S})	Egalitarian Welfare (W_{E})	NJC Fairness (F_{NJC})	Useful resource usa
Oracle-SW	0.892 ± 0.004	0.324 ± 0.008	0.336 ± 0.004	0.964 ± 0.002
Oracle-EW	0.752 ± 0.003	0.412 ± 0.007	0.272 ± 0.002	0.997 ± 0.000
Oracle-NJC	0.828 ± 0.002	0.373 ± 0.008	0.999 ± 0.000	0.991 ± 0.000
Cilantro-SW	$\mathbf{0.864 \pm 0.006}$	0.337 ± 0.013	0.513 ± 0.020	0.818 ± 0.012
Cilantro-EW	0.760 ± 0.007	$\mathbf{0.390 \pm 0.020}$	0.426 ± 0.037	$\mathbf{0.954 \pm 0.012}$
Cilantro-NJC	0.823 ± 0.002	0.355 ± 0.005	$\mathbf{0.964 \pm 0.006}$	0.931 ± 0.003
EvoAlg-SW	0.649 ± 0.017	0.131 ± 0.016	0.182 ± 0.048	0.671 ± 0.021
EvoAlg-EW	0.687 ± 0.011	0.158 ± 0.012	0.387 ± 0.040	0.700 ± 0.009
Resource-Fair	0.611 ± 0.002	0.151 ± 0.006	$\mathbf{1.000 \pm 0.000}$	0.766 ± 0.001
Greedy-EW	0.724 ± 0.005	0.306 ± 0.006	0.518 ± 0.009	0.882 ± 0.004
Ernest	0.675 ± 0.002	0.214 ± 0.005	0.891 ± 0.013	0.774 ± 0.002
Quasar	0.756 ± 0.002	0.095 ± 0.003	0.060 ± 0.003	0.706 ± 0.002
Minerva	0.555 ± 0.017	0.082 ± 0.006	0.034 ± 0.005	0.407 ± 0.023
Parties	0.661 ± 0.002	0.285 ± 0.006	0.645 ± 0.000	0.766 ± 0.001
MIAD	0.761 ± 0.002	0.285 ± 0.005	0.745 ± 0.000	0.766 ± 0.001

Table 3.2: The social welfare (3.1), egalitarian welfare (3.2), NJC fairness metric (3.3), and the effective resource usage (3.8) for all 13 methods. Higher is better for all four metrics, and the maximum and minimum possible values for all metrics are 1 and 0. The values shown in bold have achieve the highest value for the specific metric, besides the oracular policies. Resource-Fair has NJC fairness $F_{\text{NJC}} = 1$ by definition.

Evaluating Strategy-proofness. We next evaluate Cilantro policies for strategy-proofness. A policy is said to be strategy-proof if an unscrupulous user cannot increase the utility of their job by misreporting their performance metrics to the scheduling policy. For this, we repeat the same experiment set up; all jobs behave exactly as before except the `db16` job which lies about its performance by either under-reporting by a factor $\times 1/2$, or over-reporting by a factor $\times 2$. By under-reporting, the user gives the impression that more resources are required to reach its SLO; in contrast, by over-reporting, a user is deceiving the scheduler to prioritize their job as they can achieve high utility with few resources. In Fig. 3.11, we report the utilities achieved by `db16` under these untruthful behaviors. We see that for Cilantro-NJC, the job's utility does not increase when over-reporting and decreases when underreporting, leaving no incentive for the user to be untruthful. In contrast, for Cilantro-EW, a user stands to gain by under-reporting while for Cilantro-SW, they gain by over-reporting. While a theoretical study of such strategy-proofness properties is beyond the scope of this work, it is interesting to empirically observe that the strategy-proofness properties of NJC fairness policies are retained in Cilantro.

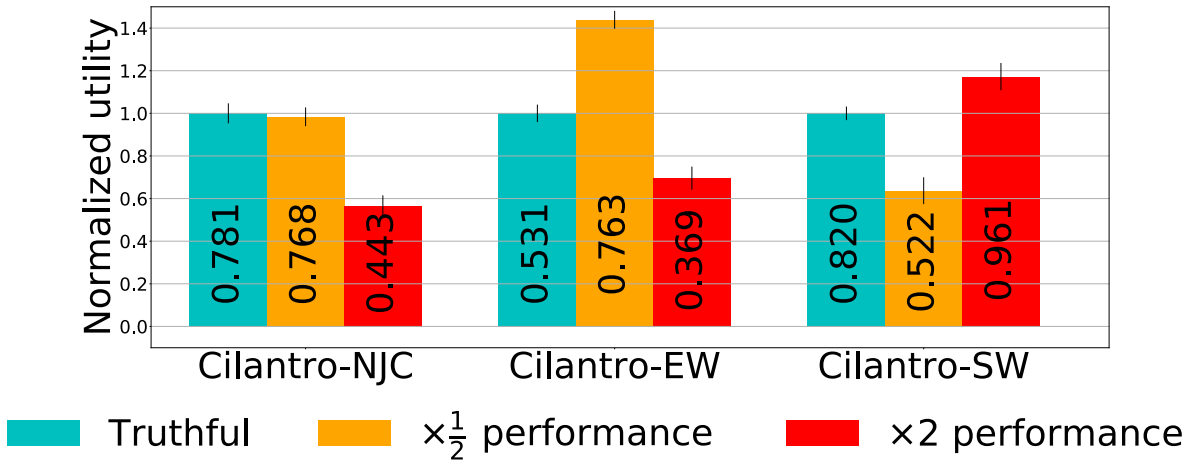


Figure 3.11: The utility of `db16` under the three online learning policies, when they report truthfully, when they under-report, and when they over-report. The plot normalizes with respect to truthful reporting, but the bars are annotated with the absolute value.

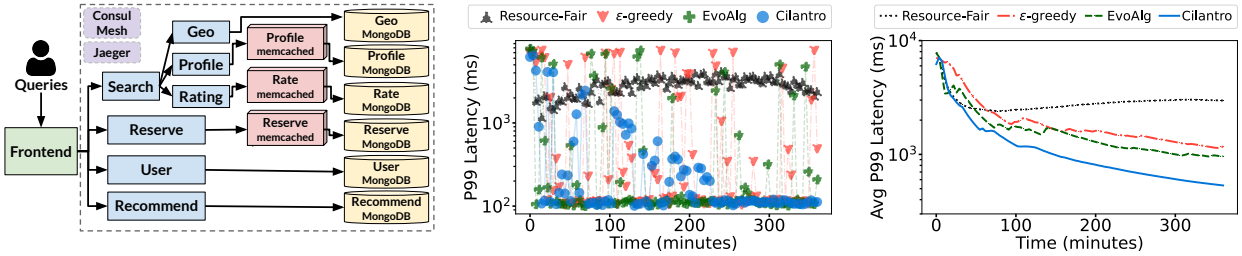


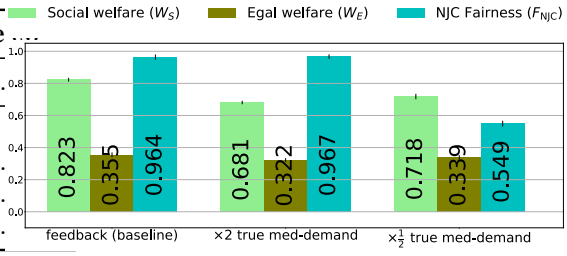
Figure 3.12: *Left:* Microservices architecture of the hotel reservation benchmark[48]. Blue boxes are business logic, red boxes are caching services, yellow boxes are databases and purple boxes are networking services. *Center:* Results for the microservices experiment comparing four methods on P99 latency over 6 hours, plotting the instantaneous P99 latency vs time. *Right:* The time-averaged P99 latency vs time.

Resource allocation for Microservices

We now demonstrate the use of Cilantro to allocate resources for inter-dependent microservices serving an application. A query to the application triggers multiple queries to different microservices and the final result is returned to the user. Cilantro must observe a single end-to-end metric, the end-to-end query latency, and then allocate fixed cluster resources to different microservices to minimize the P99 latency of the application. We note that Cilantro does not require meta information about the microservices, such as their dependency and control flow graphs; Cilantro directly optimizes the end-to-end metric as described in §3.7.

Workload. We use the Hotel Reservation application from DeathStarBench[48]. It has 19 microservices, including 6 MongoDB databases, 3 memcached kv-stores and a nginx webserver running on a consul service mesh. The architecture is shown in Fig. 3.12-Left. Collectively,

Operation	Call time
Model Update	$0.0413 \pm 0.$
`get-alloc` call	
Cilantro-SW	$2.8823 \pm 0.$
Cilantro-EW	$2.1239 \pm 0.$
Cilantro-NJC	$0.0081 \pm 0.$



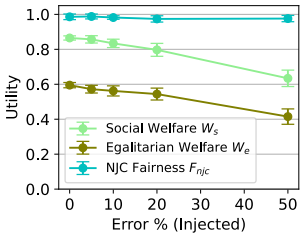


Figure 3.13: Cilantro microbenchmarks. *Left:* Mean time taken (in seconds) by Cilantro to update the performance model and for computing a new allocation for each of the three fixed cluster sharing policies. *Center:* Evaluation of Cilantro's fallback option, where users provide a demand value if they cannot report performance metrics. We evaluate Cilantro-NJC when 5 out of 20 users use this option. Since the true demand cannot be known, we use either half or twice the true demand under the median load from our profiled data. *Right:* The three performance metrics for Cilantro-NJC when we artificially introduce error to the confidence intervals of the performance and load.

these microservices serve search, recommendation, rating, account management and geolocation queries from users. We use wrk2[172] to process and submit the query workload provided in [48]. We measure the end-to-end latency of queries submitted to the frontend microservice. All microservices experiments are run on a 160 CPU cluster with 20 AWS m5.2xlarge instances.

Baselines. We compare Cilantro's end-to-end policy (§3.4) against three baselines. Resource-Fair always equally allocates the resources among microservices. EvoAlg is an evolutionary algorithm which optimizes for the P99 latency. ϵ-greedy randomly picks a new allocation with probability $1/3$, or uses the allocation with the smallest observed P99 latency with probability $2/3$.

Results and Discussion. Fig. 3.12 shows how the instantaneous and time-averaged P99 latency (computed in 30s intervals) evolves with time during the course of the experiment. Both Cilantro and EvoAlg explore early on (Fig. 3.12-Center), but as they find better values, exploration shrinks as they focus on testing more promising allocations. However, Cilantro's OFU-based online learning policy is able to do this more effectively than EvoAlg. ϵ-greedy explores aggressively even in later stages and is unable to adequately exploit good candidates it may have discovered in the early stages. Overall, Cilantro achieves a mean P99 of 525ms, compared to 930ms for EvoAlg, the next best baseline.

Microbenchmarks

Cilantro Overhead. Fig. 3.13-Left evaluates the time taken for Cilantro to process the feedback and compute the allocations for the three policies described in Sec. 3.4. This shows that Cilantro is fairly light-weight. For comparison, the average time it took to de-allocate a Kubernetes pod and assign it to a different job was on the order of 5-15s.

Unavailable performance metrics. In real-world situations, performance metrics of all users may not be available. We evaluate Cilantro's fallback defaults for such instances. We re-run the same experiment in §3.7, but for users `db01`, `mlt1`, `mlt2`, `db11`, and `prs1`, we manually set the

demand as described in §3.5. Since the true demands are not known a priori, users might under- or overstate them. To reflect this, we first compute the true demand for each user under the median load from our profiled data. We evaluate Cilantro-NJC when these five users report either half this value as their demand or twice this value, when compared to providing feedback. Fig. 3.13-Center presents results on the three criteria given in §3.4. While the fallback options are worse than when reporting feedback, the failures are graceful. Cilantro is still able to learn from the remaining 15 users and achieve efficient allocations with only relatively small drops in social and egalitarian welfare. The NJC fairness criterion is significantly small when under-reporting since these 5 users will have been allocated at most half of their true demand and F_{NJC} (3.3) depends on the single worst fairness violation.

Robustness to choice of learners and feedback errors. While Cilantro's decoupled design aids with generality, it may be susceptible to the idiosyncrasies of the specific models used for the performance learners and load forecasters. Moreover, in many real environments, the feedback can be very noisy. To show that Cilantro is robust to both these effects, we perform the following microbenchmark in a synthetic 5 user environment (described in the Appendix) with the Cilantro-NJC policy. As both feedback noise and model idiosyncrasies can be modeled with inaccurate confidence intervals, we introduce increasing levels of noise (5%, 10%, 20%, 50%) to the upper and lower confidence bounds returned by the learners and forecasters. The results, given in Fig. 3.13-Right, show that the social and egalitarian welfare decrease gracefully with noise. Moreover, due to Cilantro-NJC's conservative approach for demand recommendations, the NJC fairness metric remains relatively high despite the noise.

3.8 Conclusion

We described Cilantro, a performance-aware framework for the allocation of a finite amount of resources among competing jobs. Our motivations were: *(i)* resource allocation policies should be performance-aware and based on real-time feedback in production environments, *(ii)* schedulers should accommodate diverse allocation objectives. We designed Cilantro to address these challenges by decoupling the performance learning from the policies and informing the policies of uncertainties in performance estimates, thus enabling the realization of several performance-aware policies in multi-tenant and microservices settings.

3.9 Acknowledgements

We thank the OSDI reviewers and our shepherd, Tim Harris, for their invaluable feedback. This work is in part supported by NSF CISE Expeditions Award CCF-1730628 and gifts from Astronomer, Google, IBM, Intel, Lacework, Microsoft, Nexla, Samsung SDS, Uber, and VMware.

Chapter 4

ESCHER: Expressive Scheduling with Ephemeral Resources

As distributed applications become increasingly complex, so do their scheduling requirements. This development calls for cluster schedulers that are not only general, but also evolvable. Unfortunately, most existing cluster schedulers are not evolvable: when confronted with new requirements, they need major rewrites to support these requirements. Examples include gang-scheduling support in Kubernetes [191, 22] or task-affinity in Spark [191]. Some cluster schedulers [154, 65] expose physical resources to applications to address this. While these approaches are evolvable, they push the burden of implementing scheduling mechanisms in addition to the policies entirely to the application.

ESCHER is a cluster scheduler design that achieves both evolvability and application-level simplicity. ESCHER uses an abstraction exposed by several recent frameworks (which we call *ephemeral resources*) that lets the application express scheduling constraints as resource requirements. These requirements are then satisfied by a simple mechanism matching resource demands to available resources. We implement ESCHER on Kubernetes and Ray, and show that this abstraction can be used to express common policies offered by monolithic schedulers while allowing applications to easily create new custom policies hitherto unsupported.

4.1 Introduction

With the end of Moore's law and Dennard scaling, developers are forced to distribute their applications to process an ever growing amount of data. As a result, the past decade has seen a proliferation of new distributed frameworks [22, 118, 65] to handle a variety of workloads from big data (e.g., batch jobs, interactive query processing) to AI applications (e.g., model training and serving).

As the number of data and AI applications grows, so do their scheduling requirements. Some examples of scheduling policies are *affinity* (i.e., co-locate computation with data to avoid costly data transfers), *anti-affinity* (i.e., schedule tasks on different machines to avoid interference), and

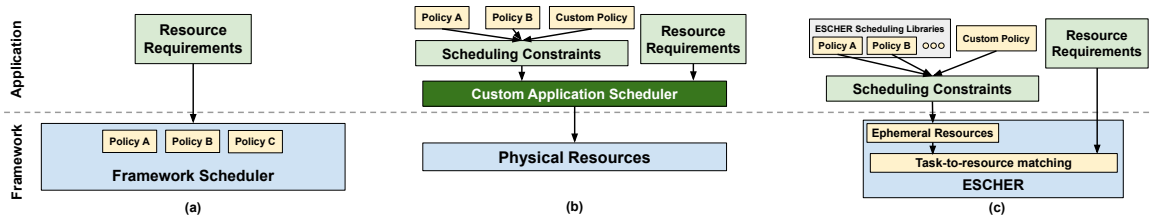


Figure 4.1: **(a)** A monolithic scheduler implements both scheduling and resource constraint matching [50, 70, 59, 22]. Some schedulers allow applications to express and compose certain policies [97, 175, 22], but custom application policies may require modifying the scheduler itself. **(b)** To maximize flexibility, some frameworks expose physical resources [65, 154], but require applications to write custom schedulers that manage both policy and resource coordination [190, 136, 35]. **(c)** ESCHER. With ephemeral resources, applications can express custom policies through ephemeral resources, while the cluster scheduler provides just one service - satisfying per-task resource constraints.

gang scheduling (i.e., schedule a group of interdependent tasks simultaneously). For example, a hyperparameter search application [95, 190] consists of multiple distributed training jobs, each consisting of multiple parallel tasks. This requires anti-affinity between jobs for high throughput, affinity within a job to avoid unnecessary data transfers, and gang scheduling to ensure multi-node jobs are not starved.

This diversity of policy requirements makes designing schedulers for distributed frameworks challenging. There is an inherent trade-off between *simplicity* and *flexibility* in exposing different policies to applications. Different cluster managers occupy different points in this trade-off space.

At one end of the spectrum (Figure 4.1a), monolithic cluster managers like YARN [179] and Kubernetes [22] provide several out-of-the-box policies for the application to chose from. This simplifies the application's task, but it compromises the flexibility, as adding a new policy requires changes to the scheduler and the cluster manager itself. Implementing a new policy requires the developer to understand and modify the source code of the cluster manager, not always an easy task given their inherent complexity. And, once the new policy is implemented, the developer is on the horns of a dilemma: either fork the project and pay the cost of maintaining it up-to-date as the project evolves, or wait many months for the change to be merged in the main branch. Worse yet, if the cluster manager is closed source, the application developer has no choice but to wait and hope that the company behind the cluster manager will implement the desired policy.

At the other end of the spectrum (Figure 4.1b), are schedulers like Omega [154] and Mesos [65] that enable an application to directly allocate resources and implement its own scheduling logic. This makes these cluster managers very flexible, but dramatically increases the complexity of the application. Implementing a scheduling policy in a distributed system can be a daunting task, as it requires not only allocating resources, but tracking the resource availability in the presence of various failures and new nodes joining the system.

In this work, we present another point in the design space that allows application developers to easily implement a range of new scheduling policies. This design point is enabled by a mechanism recently introduced by cluster managers like Kubernetes and Ray which provides an interface for

applications to dynamically create, modify, and destroy logical resources. We call these resources *ephemeral resources*. Like regular resources, ephemeral resources are pairs of labels and count values which can be allocated to tasks. The scheduler treats ephemeral resources in the same way as physical resources, subjecting them to admission control to ensure they are not oversubscribed. This frees the applications from performing admission control and tracking availability.

We find that a surprisingly large number of scheduling policies can be expressed by dynamically creating, updating and destroying ephemeral resources. Consider a simple scheduling constraint to colocate two tasks T1 and T2. To express this constraint, an application would submit T1, which creates an ephemeral resource R1 during execution, and then submit T2 with R1 as a resource requirement. The scheduler is then forced to place T2 on the same node as T1, since no other node has resource R1. While this is a very simple example, it illustrates the underutilized power of ephemeral resources for satisfying application-level scheduling constraints. In contrast, a monolithic cluster manager would have to expose a primitive designed specifically for task-task affinity, and a two-level scheduler application would have to implement the entire policy themselves, choosing where *both* T1 and T2 execute.

The key promise of ephemeral resources is that they enable an application to implement new scheduling policies *not* supported by the underlying cluster manager. This increases the velocity of deploying and iterating on new application functionality. However, there are two natural questions that follow. First, how general are the scheduling policies enabled by ephemeral resources? Second, what are the costs in terms of implementation complexity and overhead compared to natively implementing the same policy in the cluster manager? After all, if these overheads dominate, then an application developer is better off building their own scheduler.

To answer these questions, we propose a scheduling architecture for distributed applications called ESCHER [*]. In ESCHER, the application uses ephemeral resources to implement its scheduling policy instead of relying on the cluster manager's baked-in policies. The key insight of ESCHER is that a broad class of heterogeneous scheduling constraints can be cast as *ephemeral resource requirements*. The underlying scheduler simply enforces these requirements. ESCHER enables applications to implement a large number of scheduling policies by (1) dynamically creating new ephemeral resources, and (2) specifying task resource requirements on these ephemeral resources. We find that by using these two simple primitives, we are able to satisfy a large set of scheduling constraints, without requiring any changes to the core scheduler or significantly affecting application performance. For instance, gang-scheduling in ESCHER can be written in 10x fewer lines of code with less than 2x overhead in scheduling latency compared to implementing the policy natively in the core scheduler (section 4.6).

However, the flexibility of ephemeral resources does not come for free. First, it increases the application complexity compared to monolithic schedulers in which applications just need to select one of the available policies. Implementing certain policies, such as gang scheduling, requires the application to implement additional mechanisms using ephemeral resources such as ghost tasks, i.e., tasks whose sole purpose is to signal when all required resources have been allocated. Second, because ephemeral resources are created dynamically, an application must

[*]ESCHER stands for Expressive SCHeduling with Ephemeral Resources.

handle infeasible requests explicitly. For example, if a task's resource request cannot be satisfied, the task will hang and it must be explicitly terminated.

To alleviate the challenge of application complexity and provide protection against invalid resource specifications, ESCHER supports simple libraries to support common policies. We call these libraries ESCHER Scheduling Libraries (ESLs). ESLs aim to provide the best of both worlds: the simplicity of monolithic schedulers, and the flexibility of adding new scheduling policies at the application level by either extending an existing ESL or creating a new one. ESLs decouple application logic and policy by abstracting ephemeral resource management for common high-level scheduling policies, thus dramatically reducing development cost via code reuse and enabling composition of simple policies into more complex ones.

To evaluate ESCHER's performance, we implement it on both Kubernetes [22] and Ray [118] by leveraging their existing implementations for label-based scheduling, which were originally intended to represent custom physical resources rather than logical scheduling constraints. We run ESCHER on a range of applications and policies, including WordCount MapReduce with max-min fair sharing on Kubernetes as well as AlphaZero [161] and distributed model training on Ray [118]. We show that ESCHER does not impact the end-to-end performance of most applications when compared to a system that implements the same policies in the core scheduler. Meanwhile, the application can use ESCHER to express additional policies not supported by the underlying core scheduler, e.g., composing gang scheduling with affinity (Section 4.6).

Thus, ESCHER shows that one can take advantage of the ephemeral resource abstraction, whose implementation is already partially provided by some cluster managers, to express a surprisingly diverse set of scheduling policies at the application level without having to touch the core scheduler. This allows users to quickly implement new policies, as needed, to improve support for their applications.

In summary, we make the following contributions:

- ESCHER, a scheduling architecture that uses ephemeral resources to express scheduling policies without modification to the core scheduler.
- Design and implementation of a wide class of scheduling policies (§4.4) using the ephemeral resources API.
- ESLs: application-level scheduling libraries that enable applications to easily compose and re-use policies.

4.2 Motivation

Table 4.1 lists some common scheduling policies required by modern distributed applications and their off-the-shelf support across different frameworks and specialized schedulers. None of the schedulers support all policies, and many were built as a one-off solution to achieve a composition of these policies. New applications which require a new policy must find alternate methods of executing it - either by using some mechanism provided by the scheduler, such as labels, or writing their own scheduler from scratch. We now give a motivating example that is insufficiently served by existing schedulers and describe how ESCHER fills this gap.

	Framework			Scheduler		
Policy	**YARN CS [179]**	**Kubernetes [22] Core / Labels**	**Spark [194]**	**Sparrow [129]**	**Gandiva [190]**	**Gorila [121]**
Task co-location Place n tasks on the same physical machine.	✓	✓/✓	✗	✓	✓	✓
Data locality Place tasks with operands.	✓	✓/✓	✓	✓	✗	✗
Elastic Load Balancing Given an unknown number of tasks, evenly spread them out across m workers.	✓	✓/✗	✗	✓	✓	✓
Bin-packing Given an unknown number of incoming tasks, minimize the number of workers used to complete the tasks.	✓	✓/✗	✗	✗	✓	✗
Anti-affinity Given two tasks, place them on distinct nodes.	✓	✓/✓	✗	✗	✓	✗
Gang scheduling Given a set of tasks, enforce all-or-none run semantics.	✗	✗/✗	✓	✗	✗	✓
Weighted Fair Queuing[17] Given a set of tasks, enforce priority ordering.	✓	✓/✗	✓	✓	✗	✗
Soft-constraints For a priority ordering of resource constraints, schedule a task with the highest possible resource satisfiability.	✗	✓/✗	✗	✗	✗	✗

Table 4.1: Common scheduling policies and off-the-shelf support from existing schedulers. Kubernetes comparision includes both modes of operation, using just the core scheduling functionality and using labels. In addition to these policies, ephemeral resources allow applications to specify and compose custom policies.

Existing systems are hard to evolve

As applications become increasingly diverse, the cluster schedulers must evolve to support novel scheduling policies required by these applications. Consider distributed training, which has emerged as a dominant ML workload. Multiple training jobs are often scheduled simultaneously due to multi-tenancy and individual users submitting multiple training runs in parallel to evaluate different model architectures. This requires several distinct scheduling policies:

- *Anti-affinity:* Evenly spread training jobs across the cluster to ensure high throughput.

- *Affinity:* Co-locate all tasks of the same training job on the same machine to avoid unnecessary data transfers.
- *Gang scheduling:* For multi-node jobs, schedule all tasks of the job simultaneously to avoid starvation.
- *Bin packing (dynamic):* Monitor job utilization and consolidate jobs to reduce resource fragmentation.

Many popular schedulers implement at least one of these policies, but it is rare for them to support all four (Table 4.1), never mind a composition of the policies. There are two fundamental challenges that make it difficult or infeasible to extend monolithic schedulers (Figure 4.1a) in this way. We use Kubernetes [22] to illustrate these challenges.

First, the application must *express* its policy using the API chosen by the framework scheduler. While some schedulers support composition, it is difficult in general to design a scheduler API that can capture all possible use cases. For example, in Kubernetes, applications specify scheduling policies with *static weights* to resolve conflicts. This can be used to express a composition of two policies that, for instance, weights affinity over anti-affinity. However, the complexity of composition is not linear in the number of policies. E.g., to add bin packing and prioritize it, the application would have to ensure that the weight of bin packing is always greater than the sum of weights for affinity and anti-affinity.

Second, the scheduler *implementation* must extend to new policies. This is difficult because the scheduler must ensure that each new policy interfaces correctly with all other existing policies. Adding another policy requires modifying Kubernetes itself, which takes significant time and effort. Dynamic policies are even more difficult to support if the scheduler was not initially designed for it. For instance, adding gang scheduling support in Kubernetes took months of discussions and the eventual feature was not mainlined and instead implemented in an add-on scheduler [109, 103, 104]. Similarly, Machine Learning pipelines involve multiple interdependent tasks (e.g., data pre-processing, training, serving) defined in a DAG. Scheduling DAGs is not natively supported in Kubernetes, leading to the emergence of specialized plugins such as Kubeflow [10].

Due to these limitations, applications must write custom schedulers to maximize performance, as Gandiva [190] did for distributed training. Unfortunately, this design requires the application to implement both the policy and the scheduler *mechanism*, maintaining resource state and handlers for task and resource management, coordination, node addition and deletion, etc. (Figure 4.1b). This is both difficult to build and extend. E.g., Gandiva (built on Kubernetes) supports affinity, anti-affinity, and dynamic bin-packing, but the addition of gang scheduling would greatly increase the complexity of the scheduler code. Thus, Gandiva remains limited to distributed training jobs that fit on a single multi-GPU node.

Static labels are insufficient

Some frameworks [65, 154, 179, 22] already support *static* label creation as string key-value pairs (e.g., "v100 GPU": 1) associated with cluster nodes. This allows cluster operators to tag nodes with physical resource attributes (e.g., CPU/GPU architecture, rack affinity) at cluster launch time, which can be requested by applications at execution time.

In ESCHER, we propose repurposing this API to express *custom application scheduling policies*, in addition to physical resource requirements. Unlike physical resources, which can be statically determined at a node's launch time, logical scheduling constraints may depend on run-time information. Therefore, it is natural to extend existing static label creation APIs to *ephemeral resources* that are dynamically created.

For example, to express task-task affinity between tasks T_1 and T_2, we must first learn where T_1 was placed before deciding the placement constraint for T_2. This can be easily done through ephemeral resources: T_1 dynamically creates a logical resource that is required by T_2. With static labels, the only option is for the application to pin T_1 and T_2 to a predetermined node.

For the same reason, there are some inter-task constraints that are fundamentally impossible to implement with static labels, such as scheduling policies that depend on *time*. One example is a DAG scheduling policy. At its core, this requires a primitive that guarantees that some task T_2 will not run until another task T_1 finishes. This is impossible to express using static labels alone, which cannot reason about the temporal ordering between two tasks.

4.3 ESCHER Design and Workflow

A scheduling policy is defined by a set of temporal and spatial dependencies between tasks and nodes. We call these dependencies *scheduling constraints*. The key idea in ESCHER is to map these scheduling constraints to resource requirements by introducing a new resource type, *ephemeral resources*.

An ephemeral resource is a logical (i.e., non-physical) resource attribute that the application can dynamically associate with a node. Like physical resources, ephemeral resources have an associated capacity and can be acquired and released by tasks. We call these resources *ephemeral* because the application can create, modify, and destroy them at runtime.

ESCHER uses ephemeral resources for implementing scheduling constraints by leveraging a common functionality provided by cluster schedulers: matching the application-specified resource requirements with the cluster's resource availability. For instance, if an application task requires two GPUs, the scheduler should schedule that task on a node that has at least two available GPUs. With this resource-matching capability, the scheduler treats an ephemeral resource like a physical resource and aims to satisfy its capacity constraints.

The implementation of scheduling policies in ESCHER follows a two-step pattern. First, the application creates ephemeral resources or updates capacities of existing ephemeral resources. This is done programmatically at runtime through the ephemeral resource API. Second, the application associates ephemeral resource requirements with tasks. Note that these steps can happen in any order. This allows the application to make targeted placement decisions (Figure 4.2) to satisfy the policy's scheduling constraints.

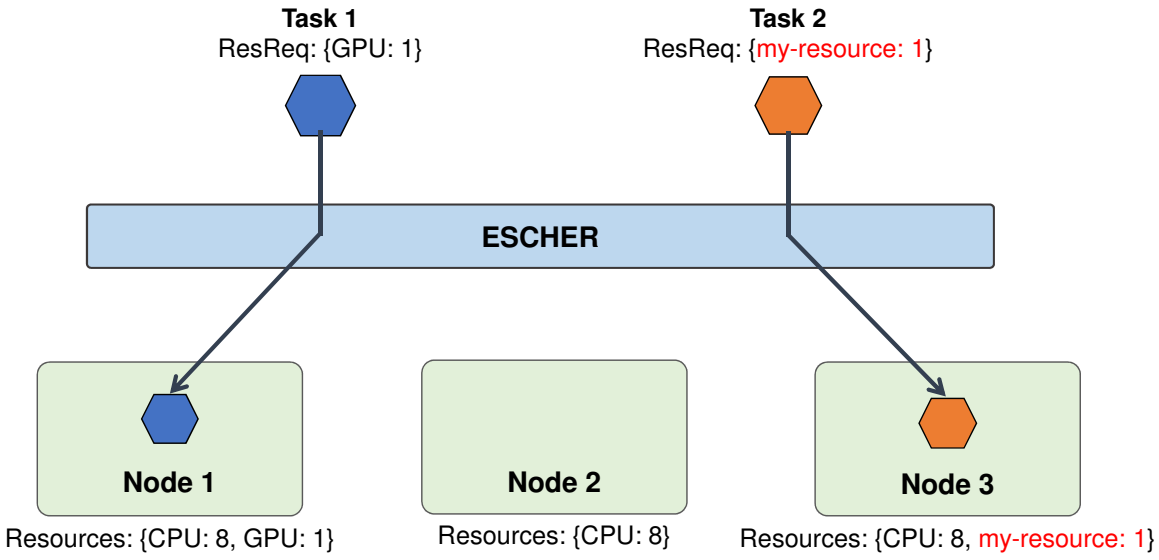


Figure 4.2: Example using ephemeral resources for task placement. Applications create ephemeral resources (*my-resource*) on the nodes where they wish to place a task and then launch a task requesting *my-resource*. The resource-matching scheduler ensures the task is placed on the desired node.

ESCHER Workflow

Figure 4.3 describes the workings of an ESCHER scheduler. ESCHER functionality, by design, is split between the application and the resource management framework. The application can specify scheduling policies through the ephemeral resource API (Section 4.3), while the framework performs resource matching and accounting over a set of underlying physical resources. We envision that most applications would specify and compose policies through the higher-level ESCHER Scheduling Library (ESL) interface, which uses the ephemeral resource API to encapsulate common policies.

When using ESLs, an interaction with the system typically starts with an application requesting a scheduling policy from the ESL (fig. 4.3). The ESL may interact with the resource manager, e.g., by reading cluster state, and implements the policy by creating the appropriate ephemeral resources. The application then receives a resource specification R from the ESL. The application attaches R to a task and submits it to the resource manager for placement.

Ephemeral Resource API

In ESCHER, the resource management framework exposes two simple API calls to manage ephemeral resources: `set_resource` and `get_cluster_status` (Listing 1). Once created, an ephemeral resource behaves as any regular physical resource and can be acquired and released by tasks.

In addition to the required parameters resource label and capacity, the `set_resource` call also allows the specification of constraints `node_spec` where the resource must be created. If

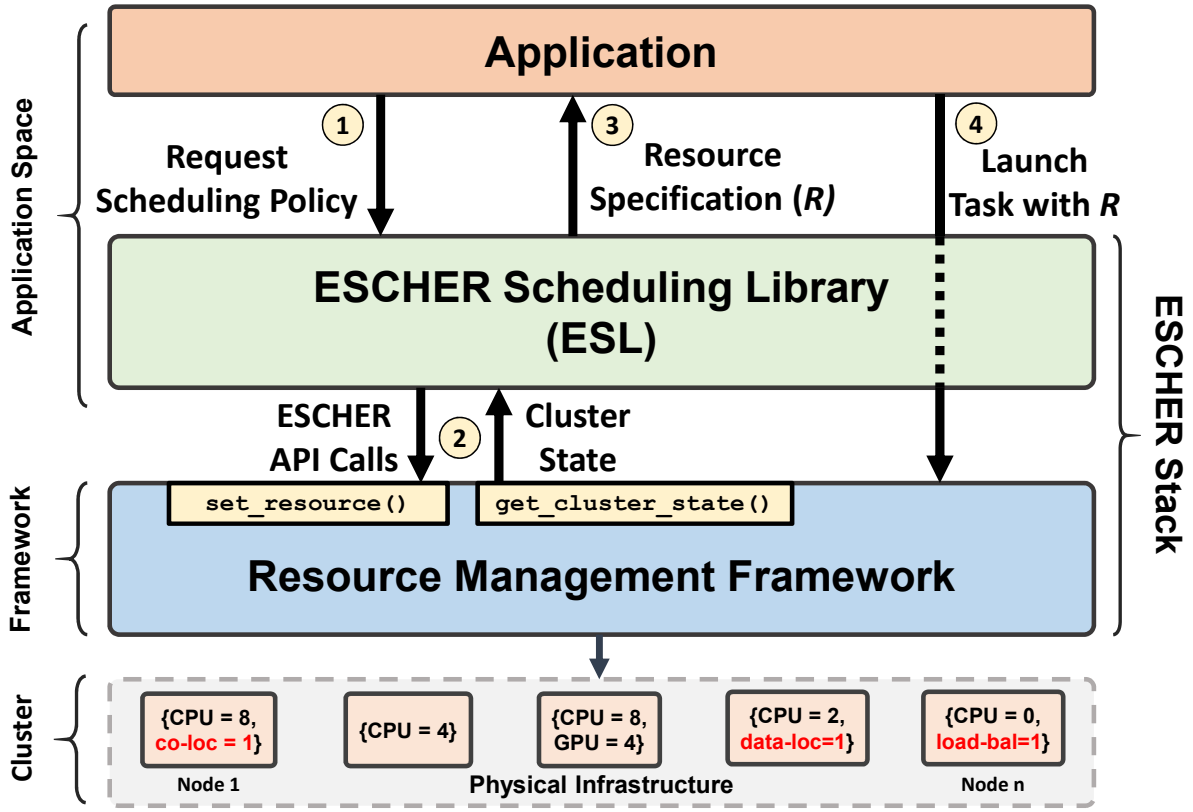


Figure 4.3: ESCHER task submission workflow with an ESL mediating the implementation. A task requests a supported scheduling policy from the ESL, which invokes the ESCHER API if necessary and returns the resource specification which would satisfy the policy. The task is launched with the returned resource specification.

`node_spec` is a resource vector, the resource is updated on all nodes where the constraint resource vector is a subset of the node's available resource vector. Optionally, a `num_nodes` field in the `node_spec` can specify how many nodes to execute `set_resource` on if multiple nodes satisfy the `node_spec` constraints. To make targeted `set_resource` calls, the `node_spec` can contain a unique node identifier (e.g., IP address).

The `get_cluster_status` call returns a mapping of node to local resource capacity and availability. These are not required for all policies, but can be useful to handle node additions and removals (Section 4.3).

The ESCHER scheduler's responsibility is to provide the minimal guarantees provided by any resource-matching scheduler: (1) A task whose resource requirements can be met by a node in the cluster will eventually be scheduled, and (2) A node is never allocated past its capacity. Together, these imply that the scheduler implements: (1) task queuing and dispatch, (2) node selection for each task, and (3) resource allocation for each task. Note that the scheduler does not need to satisfy any other constraints, such as a promise regarding the node where a task is actually scheduled.

```
# Create resource with name and capacity on a node
# where the 'node_spec' constraints are satisfied.
# node_spec can be a resource vector or node id.
# If node_spec = None, resource is set locally
def set_resource(name, capacity, node_spec)

# Returns cluster resource state as a list of
# node-wise map of local resource availability.
def get_cluster_status()
```

Listing 1: Ephemeral resource API

ESLs - ESCHER Scheduling Libraries

Controlling task placement through direct manipulation of ephemeral resources can be burdensome for applications with conventional scheduling requirements. To reduce the application complexity and delineate scheduling policies from the mechanisms to implement them, we propose ESCHER Scheduling Libraries (ESLs).

The role of an ESL is simple - given a set of tasks, generate and apply a set of ephemeral resource requirements on the tasks which satisfy the desired scheduling policy. ESLs achieve this by encapsulating the state management for ephemeral resources and providing a unified API for implementing domain-specific scheduling policies. An application requests a scheduling policy supported by an ESL, which then makes the appropriate ESCHER API calls and returns the resource specification the application must use to realize its desired policy. In our implementations, ESLs are designed as a daemon that can service scheduling requests made from a single or multiple applications.

ESLs are similar in spirit to Library Operating Systems (LibOS [78]) in the Exokernel [43] design. In the same way that a LibOS encapsulates the complexity of direct resource management exposed by an Exokernel, an ESL abstracts the policy implementation and enables sharing across different applications. Moreover, this design makes applications portable across different cluster frameworks, e.g., Ray vs. Kubernetes (Section 4.5), since ESLs separate the application policy from the application code. Finally, ESLs can protect applications from invalid specification of ephemeral resources by validating resource requests before launching tasks.

Since distributed applications can have widely varying scheduling requirements, we anticipate the development of domain-specific ESLs which can strike a balance between generality and preserving domain-specific optimizations. As an example, we describe and implement an ESL for hierarchical max-min fair sharing in Section 4.4 and Section 4.6.

Fault tolerance

In the event of a node failure, ESCHER works in tandem with the fault-tolerance scheme of the underlying framework. Most frameworks [118, 22] simply re-execute the tasks with the same

resource requirement. However, since these resource requests include ephemeral resources which no longer exist, these re-executed tasks cannot be scheduled.

At the bare minimum, an ESL must ensure that ephemeral resources are restored at some other eligible node. To do this, ESCHER relies on the cluster framework to report failed nodes through the `get_cluster_status` API. Once failed nodes are detected, an ESL can recreate the ephemeral resources on suitable nodes by replaying the `set_resource` calls for the failed resources. If a candidate node is found, the resource is recreated, else the tasks must wait for the failed node to recover before they can be rescheduled. When a node is restored, its resources are also reinitialized, allowing any waiting tasks to get rescheduled.

Evolvability and complexity in ESCHER

The ability to create ephemeral resources on targeted nodes makes scheduling with ESCHER as flexible as letting the application directly control task placement on cluster nodes. Indeed, scheduling a task T on node N is equivalent to assigning a uniquely-named ephemeral resource R_N with capacity 1 to node N, and having T request one unit of resource R_N.

While this targeted placement makes ESCHER highly evolvable, what are its benefits over simply yielding placement control over resources to the application directly? After all, if the goal is to schedule a task on a particular node, ESCHER makes this operation arguably more complex as it requires creating an ephemeral resource on the node.

The primary benefit of ESCHER is that policy and ESL implementations do not need to reason about *tasks*. In a framework with fully application-level scheduling, such as Mesos or Omega [65, 154], the application scheduler has to maintain possibly distributed state about the current set of tasks. When a task is submitted and can't yet be scheduled, the scheduler queues the task. When a task starts, the scheduler must update the current resource availability to ensure that resources do not become oversubscribed. On task completion, the scheduler must again update the current resource availability and select a new task to run from the queue.

Of course, none of these functionalities are unique to ESCHER. Task to resource matching is a necessity to every scheduler system, which is why it is the core responsibility that we assign to the ESCHER scheduler. Thus, in ESCHER, *an application that has no specific policy requirements can use an ESCHER scheduler directly without implementing any scheduling code*. This is not possible in systems that expose resources directly to the application, such as Mesos [65] or Omega [154], as it is expected that the application will also implement all mechanisms related to task scheduling.

For applications that do require a custom policy or an ESL, this division of responsibilities still reduces the effort required from the application developer, compared to implementing a complete scheduler. For example, most of the policies that we present in Table 4.2 do not require examining the current cluster state; it is enough for a task to create a resource on its local node or create resources on all nodes that match a given resource spec. The exceptions are gang scheduling, which requires reading the cluster state to roll back a group of tasks in case of a failure, and load-balancing, which computes the current load from the cluster state. In contrast, a fully application-level scheduler must continually update and reason about the current cluster state in order to find a

Policy Example	Primitive used	Implementation with ESCHER
Sequential: Run T_2 after T_1.	Signaling	T_2 requests ephemeral resource E created by T_1 on completion.
Gang Scheduling: Run T_1 and T_2 simultaneously.	Locking, Signaling	Two ghost tasks T_1^g and T_2^g request 1 CPU each, and each creates an ephemeral resource E_{CPU} of capacity one. When both T_1^g and T_2^g run, schedule T_1 and T_2, each requesting one unit of E_{CPU}.
Affinity: Run T_1 and T_2 on the same node.	Locality	A ghost task T^g requests 2 CPUs, and creates an ephemeral resource E with capacity 2. T_1 and T_2 request one unit of E each.
Anti-affinity: Run T_1 and T_2 on different nodes.	Queues	Every node in the cluster creates anti-affinity resource E with capacity 1. T_1 and T_2 request one unit of E each.[1]
Load-balancing: Evenly spread tasks across nodes.	Queues	Create load-balancing resource L with capacity 1 on each node. Each task requests one unit of L each. When all L resources are exhausted, increase capacity by 1.
Data-locality: Run T where its input D is stored.	Locality	When storing D, create ephemeral resource E_D on the same node. T requests E_D.

[1] If T_1 and T_2 are long-running, the application cannot use the nodes they are running on for other anti-affinity placements. To avoid this, we have two short-lived ghost tasks T_1^g and T_2^g request 1 unit of E each, create ephemeral resources E_1 and E_2, and then terminate. T_1 requests E_1 and T_2 requests E_2.

Table 4.2: Expressing scheduling constraints with ephemeral resources

suitable node for each task. In the ESCHER design, this responsibility is given to the system rather than the application.

4.4 Scheduling with ESCHER

A scheduling policy is a mapping of tasks to resources which satisfies any spatial ("where") and temporal ("when") constraints. We now describe how these constraints can be cast as resource requirements with ESCHER.

Scheduling primitives in ESCHER

We present four scheduling primitives implemented using ephemeral resources which can be used to express both spatial and temporal constraints. We note that this is not an exhaustive set of primitives possible with ephemeral resources. Applications have the flexibility to define their own primitives through ephemeral resource manipulation.

[P1] Locality. Tasks must often be co-scheduled on the same physical node as another task or must be co-located with data. These spatial constraints can be easily expressed in ESCHER. The target task for co-location creates a local ephemeral resource E_r with unbounded capacity when

the task starts or the data to be co-located with is created. The constrained task then requests 1 unit of E_r and, thus, automatically gets scheduled on the same node.

[P2] Task Signaling. Distributed applications rely on expensive RPCs to coordinate the execution of interdependent tasks. This is prevalent in directed acyclic graph (DAG) task schedulers, where the ordering of tasks is critical for correctness. These temporal constraints can be expressed with ESCHER by creating ephemeral resources dynamically, effectively using them as signals. E.g., if task T2 has *any* "happens-before" dependency on task T1, T1 can create a resource E_{T2} when it completes. T2 *a priori* requests E_{T2} as a part of its resource requirements when launched, and thus is scheduled as soon as E_{T2} is created by T1. Note that signals in ESCHER are single-shot—all task requests are declaratively placed at the start, and tasks begin execution only when their ephemeral resource demands are met by newly created resources. More generally, barrier synchronization is naturally supported. Given $\left\{T_j^{i-1}\right\} \rightarrow T^i$ for $j > 1$, T^i could simply request a single unit of resource created by each of T_j^{i-1} upon their respective completion. Thus, semaphores (and therefore, mutual exclusion) can also be implemented.

[P3] Queues. Many policies [17, 24] use one or more task queues as a fundamental construct in their implementation. The core ESCHER scheduler queues tasks until their resource requirements (ephemeral and physical) can be satisfied. ESCHER allows the application to decide when to dequeue tasks by increasing the capacity of an ephemeral resource. Creating a queue is simply creating a unique ephemeral resource E_q with initial capacity 0 on any node. A task is enqueued by launching it in a wrapper task requesting 1 unit of E_q resource. The queue drain rate can be set by changing the capacity of E_q. On acquiring the E_q resource, the wrapper task submits the contained task to the scheduler with its physical resource requirement (e.g., 2 CPUs) and exits. Note that it's possible to implement batched scheduling by incrementing the capacity of E_q by the desired batch size.

[P4] Resource Locking. ESCHER enables a new scheduling construct where an ephemeral resource can be used to acquire and lock one or more physical resources ("bundle"). This reservation of resources is achieved with *ghost tasks* - long-running tasks which acquire the bundle like a regular task and create a local ephemeral resource to accommodate new tasks. Ghost tasks create a pattern of indirection where tasks request ephemeral resources instead of physical resources to get scheduled. We note that ghost tasks achieve the same outcome as incremental locking presented in Omega [154]. This is useful when applications require atomic transactions on a *group* of resources, such as in gang scheduling.

Scheduling policies with ESCHER

To illustrate the use of these scheduling primitive constructs, we now describe the implementation of an example application-level policy and a cluster-level policy with ESCHER. Table 4.2 lists more policies and their implementation with ESCHER.

```
def soft_constraint_scheduler(task,
↪  ordinal_resource_preferences):
  for res_pref in
  ↪  ordinal_resource_preferences:
    if recv_heartbeat(task.task_id):
      break
    task.resources = res_pref
    task.launch()
    sleep(timeout)

def main():
  res_prefs = [{gpu: 1}, {cpu: 1}]
  soft_constraint_scheduler(task, res_prefs)
```

(a)

```
def task1(id):
  set_resource(label=id, capacity=1)
  ...
def composite_scheduling():
  # Create load-balancing resources
  for node in cluster:
    set_resource("load_balancing", 1, node)
  for i in range(0, task_count):
    # Load balance task 1
    task1.launch(id=i, resources =
    ↪  {'load_balancing': 1})
    # Co-locate task 1 & 2
    task2.launch(resources = {i: 1})
```

(b)

Figure 4.4: Scheduling with ESCHER. **(a)** Soft constraints with ESCHER. **(b)** Composition of load-balancing and co-location policies with ephemeral resources in ESCHER.

Application Policy: Gang Scheduling

Distributed training [190] and reinforcement learning workloads [121] require gang scheduling, where all tasks should start and run concurrently. This implies all-or-none scheduling semantics, where either all resources requested by all tasks are granted simultaneously, or no resources are granted.

In implementing all-or-none constraints, a common requirement is to check whether sufficient resources are available to satisfy the policy and reserving them, if necessary. To achieve this, ESCHER uses *ghost tasks* from the resource locking primitive. For instance, gang-scheduling a pool of 8 tasks (each of which requires 1 CPU) can be done by launching a ghost task which requires 8 CPUs and creates 8 units of `gang-sched` resource. If all ghost tasks are successful, each task in the pool can then request 1 `gang-sched` resource and 0 CPUs to get scheduled. If any ghost task is unsuccessful, a timeout in other ghost tasks executes a rollback and removes the `gang-sched` resource. To avoid live-locks, either the applications can execute an exponential back-off [171] before retrying, or an ESL can serialize all gang scheduling requests through a common shared library. We discuss this design space in Section 4.6.

Cluster Policy: Hierarchical Fair Sharing

From a cluster operator's perspective, using ESCHER allows enforcement of cluster-level scheduling goals, such as multi-tenancy, while still supporting application-level scheduling policies described above. For example, consider large organizations, which typically have a cluster of resources shared among teams. This sharing has three requirements. First, the scheduler must allow assigning resource sharing weights to users. Second, to maximize resource utilization, the scheduler must implement max-min fairness [50], i.e., temporarily re-allocate idle resources to oversubscribed users. Finally, teams need to further partition their share of resources among sub-teams.

Hierarchical max-min fair sharing (HFS) can be implemented as an ESL using a variant of the Queue primitive. The HFS ESL is instantiated to operate on a specified domain of nodes and provides a single call - `create_user(id, weight)` - which returns a resource name unique to

the user id. On invoking this routine, the ESL executes a `set_resource` call to create a unique resource (e.g., `res_user1`) with infinite capacity for the user on each allocated node. When a user submits a task, they must request a capacity of 1 their unique resource label (e.g., `res_user1`) which ensures their task is run only on the resources provisioned for them. Since ephemeral resources can be updated at runtime, the ESL dynamically resizes user allocations by adding and removing their ephemeral resources. Hierarchies in this setup can be created by launching multiple instances of the ESL and restricting their operating domain to the nodes granted by the parent ESL.

Soft constraints with ESCHER

The core scheduler enforces task resource requirements as a hard constraint, keeping the core scheduling logic simple. However, some applications may demand relaxed scheduling semantics, where some resource requirements can be specified as *soft constraints*. Ephemeral resources can be used to implement soft constraints even when the scheduler only supports strict matching of resource requirements. First, the application specifies the soft constraints as an ordinal set of resource set preferences $R = [r_1, r_2, ..., r_n]$, where r_i is the i^{th} preferred resource requirement set. For instance, an application which prefers a GPU but will work without one would specify its resource requirements as $R = [\{gpu : 1\}, \{\}]$.

The soft-constraints ESL then instruments the application's tasks with a lightweight heartbeat sent to the ESL to notify it of successful scheduling when the task launches (Listing 4.4a). The ESL then sequentially attempts to launch a task, starting with resource requirement r_1. If the ESL does not receive the callback from the task within a certain timeout t, it implies the resource requirement was not matched. The ESL then cancels and resubmits the task, now with a resource requirement r_2. This best-effort scheduling is attempted for all resource preferences $r \in R$ until the scheduling succeeds or all preferences have been evaluated.

Objective functions. Soft constraints can be used to approximate policies that optimize a combined objective function. For instance, consider a policy which aims to balance both CPU and memory utilization in a cluster. Formally, given weights for memory and CPU utilization α and β, the objective is to maximize the minimum of $\gamma_n = \alpha M_n + \beta C_n$ across all nodes, where M_n and C_n are the memory and CPU utilization on node n (ranging between 0 and 1).

To express this policy in ESCHER, the scheduler first creates the resource *obj* on each node with a capacity of $\alpha + \beta$. A task with memory and CPU requirements m and n then specifies the soft constraint $R = [\{obj : \gamma\}, \{obj : \frac{\gamma}{2}\}, \{obj : \frac{\gamma}{4}\}, ...\{obj : \frac{\gamma}{2^k}\}]$, where $\gamma = \alpha m + \beta n$. The task also includes the hard constraints $\{MEM : m, CPU : n\}$. This preference places each task on the least utilized node, correct up to a factor of 2, while guaranteeing that no node is overallocated.

Policy Composition

Applications like hyperparameter search require a hierarchical composition of other policies (Section 4.2). Scheduling policy compositions can be logically expressed either as an AND conjunction or an OR conjunction. AND constraints are expressed by concatenating the ephemeral resource vectors for two policies. OR constraints are supported using soft constraints. For instance, if an ap-

plication wants to co-locate $task1$ and $task2$ while load balancing their groups across the cluster, it can simply apply co-location on $task1$ and $task2$ and load balancing only on $task1$ as shown in fig. 4.4b. Similarly, cluster-level policies can be composed with application-level policies (Section 4.6).

Conflicting compositions of policies may render a task impossible to schedule. E.g., composing a cluster-level fair sharing policy and an anti-affinity policy may result in a infeasible task if the fair share of resources is insufficient. In such situations, one can specify conflict resolutions by using soft-constraints to relax scheduling policies. For instance, a soft constraint vector [{fair_a: 1, anti_aff: 1}, {fair_a: 1}] would try scheduling a task with fairness and anti-affinity, and relax the anti-affinity constraint if a conflict arises.

4.5 Implementation

ESCHER is a design that can be applied to both centralized and decentralized schedulers. We built two prototypes of ESCHER, on Kubernetes [22], a container orchestration framework with a centralized scheduler, and Ray [118], a distributed execution framework with a decentralized scheduler.

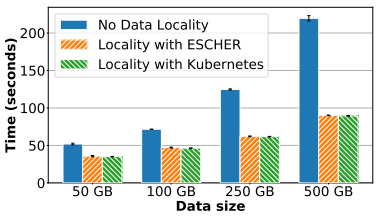
ESCHER inherits the scheduling properties of the parent cluster framework. For instance, it can utilize fractional and heterogeneous resources on Ray and Kubernetes. Since Ray and Kubernetes have a task-by-task scheduler, our current implementation also schedules in a greedy, task-by-task manner. However, ESCHER's queuing primitive can be used to extend a task-by-task scheduler to do batch scheduling.

Each framework handles isolation differently, which affects how ghost tasks are implemented. Ray does not enforce CPU affinity, so a ghost task can block the logical resource for the actual task without blocking the physical CPU. Kubernetes enforces isolation through cgroups. In this case, the ghost task reserves the CPU for its cgroup and when the actual task is scheduled, it is added to the same cgroup by running cgclassify in the task's preamble. The source code is available at https://github.com/romilbhardwaj/escher.

Kubernetes Implementation. A Kubernetes task (pod) specifies its constraints in the form of two sets: a set of filtering policies, such as resource demands, to enforce hard constraints and a set of weights for these built-in policies to add soft constraints. The scheduler first finds a set of candidate nodes, then computes a policy-weighted score for each node to find the best fit.

Kubernetes also allows the definition of arbitrary string and integer pairs associated with nodes known as *extended resources*. Extended resources are identical to regular resources in that they can be acquired and released by Pods, except they can be defined as arbitrary key value pairs. The extended resources API has conventionally been used by cluster operators for marking specialized hardware as a one-time operation when the cluster is initialized. In fact, application pods are not granted access to this API by default.

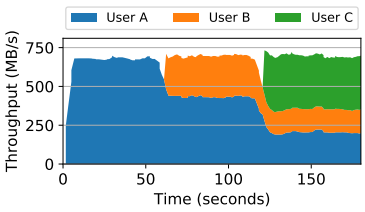
To implement the ESCHER *set_resource* API, we change the Kubernetes role-based access control to allow applications to directly invoke the extended resources API in Kubernetes. This grants applications the ability to create and update *extended resources* directly using the *patch_node_status*



(a)

Nodes	Scheduler: Generic	Scheduler: Kubernetes
10	183.32 ± 0.51	54.69 ± 0.46
50	113.71 ± 0.49	44.02 ± 0.27
100	51.90 ± 0.31	35.08 ± 0.31

(b)



(c)

Figure 4.5: Data locality and hierarchical max-min fair sharing for WordCount. **(a)** Makespan of WordCount running on a 100-node Kubernetes cluster, comparing a random placement policy, ESCHER on Kubernetes with data locality, and Kubernetes' native data locality. **(b)** Makespan of WordCount MapReduce jobs in seconds across varying cluster sizes. **(c)** Hierarchical max-min fair sharing with ESCHER. A and B are in Sub-Org1 with weights 2:3; C is in Sub-Org2. A, B, and C begin submitting tasks at t =0, 60, and 120, respectively.

call. From a security perspective, ESCHER applications require access only to create and remove extended resources. The Kubernetes API should deny write access to physical resources. Additionally, we employ namespacing of resources to enforce isolation and prevent malicious applications from overriding other applications.

To force the Kubernetes scheduler to act as a simple resource-matching scheduler, the scheduler is invoked with only hard resource constraints set in the filtering policy. Weights for all other policies are set to zero. Thus, the combination of *extended resources* API with hard resource constraints makes it possible to implement ESCHER policies on Kubernetes *without making any changes to the core Kubernetes scheduler*.

Ray Implementation. Ray runs a scheduler per node, which collectively implement a bottom-up decentralized resource matching mechanism [118]. Each node in the cluster has a set of key-value pairs signifying resource labels and their quantity, e.g., `{"cpu": 8, "gpu": 1}`. Each task specifies its hard resource requirements with the same data structure.

Ray nodes share a centralized log of the total resource capacity at each node, where each entry represents the capacity at a node. The Ray scheduler matches a task to resources by storing resource availabilities from other nodes in a map and allocating the first node which can satisfy the task's resource request. Since a scheduler's view of the global state is only eventually consistent, it can correct previous decisions by running the same logic to find another eligible node. To support ESCHER, we extend the Ray core to permit updates to each node's resource capacity at runtime. We restructure Ray's centralized log so that each entry stores only a *delta*, instead of the absolute capacity at that node. This guarantees no race conditions occur between resource updates, while avoiding expensive coordination, e.g., via a distributed lock.

4.6 Evaluation

In this section, we evaluate the following questions:

- Can existing distributed applications be ported to use ESCHER and what are its implications?
- What are the tradeoffs with implementing scheduling policies in the application space vs in the framework?
- What are the overheads of scheduling with ephemeral resources?

All evaluations use Amazon EC2 m5.12xlarge, m5.4xlarge or p3.8xlarge instances. Kubernetes clusters are provisioned using Amazon EKS running version 1.19.

End-to-end Evaluation

WordCount with MapReduce

WordCount counts the number of words in large text datasets and is often implemented with MapReduce [35]. To avoid expensive data transfers, data locality is essential. We implement the map and reduce tasks as independent operators running in containers. The input files are chunks of a file with random words, each hosted by one of 100 nodes. The total input size is varied from 50 GB to 500 GB. We implement ESCHER on Kubernetes, using an ESL for data locality (Section 4.4), and compare against Kubernetes's built-in data-locality policy [11] and a locality-unaware random policy.

Unsurprisingly, Figure 4.5a shows that as the input size increases, the overhead of transferring chunks over the network dominates the mapper computation time for the no-locality policy, taking up to 58.3% of the total job time when the input size is 500 GB. Meanwhile, ESCHER on Kubernetes provides the same performance as Kubernetes itself, but without modifying the core scheduler framework. Furthermore, Figure 4.5b shows that ESCHER can also scale with the cluster size. Throughout different scales, ESCHER performs comparably with the core Kubernetes scheduler, with its makespan staying within 1.9% of the baseline Kubernetes scheduler.. Implementing data locality with ESCHER required adding only two lines: a `set_resource` call during data generation to create a local `data-<id>` resource and a line to specify a `data-<id>` resource requirement for the mapper tasks.

Hierarchical max-min fair sharing

Hierarchical Max-Min Fair Sharing (Section 4.4) allocates resources proportionate to a user's weight in a hierarchical organization. Users submit jobs at different times, so their ideal absolute resource share is dynamic, making it impossible to maximize overall resource utilization with static labels. For example, consider a two-team organization: Sub-Org1 with users A and B of weights 2:3, and Sub-Org2 with user C. To ensure fairness with static labels, the only option is to allocate each user a fixed proportionate share, leading to under-utilization when only one user is submitting work.

Because ephemeral resources can be *dynamically* created and destroyed, an HFS policy ensures fairness while also maximizing overall utilization as users enter and leave the system (Figure 4.5c). We deploy a HFS policy on a 100 node cluster running WordCount. We use a parent ESL for the

teams and two children ESLs for Sub-Orgs 1 and 2 to create a hierarchy of ESLs. An HFS ESL tracks idle resources and reallocates resources between teams or users. The workload in Figure 4.5c starts with only user A submitting tasks to the scheduler. Since other users' resources are idle, the HFS ESLs re-allocate all idle resources to A to achieve max-min fairness. At time t=60, user B starts submitting tasks. This causes the Sub-Org 1 ESL to reclaim resources from A to re-allocate to B, in proportion to their weights. B's warmup time causes a small dip in net throughput at t=60. Finally at time t=120, user C starts submitting tasks and the parent ESL reallocates resources to Sub-Org2. Since Sub-Org 2 and Sub-Org 1 have equal weights, C's resource allocation is equal to the sum of A and B's allocation. Ephemeral resources also enable composition: the application composes its custom policy (in this case, data locality for WordCount) with the two HFS ESLs by concatenating all the resource requirements.

AlphaZero

AlphaZero [162] is a reinforcement learning application for the board game Go. We demonstrate ESCHER's flexibility by porting an implementation [8] onto Ray without compromising performance relative to the optimal hard-coded (but inflexible) placement.

AlphaZero executes a Monte Carlo Tree Search on the game state space in a CPU-intensive *BoardAggregator* process. The search is guided by a *PredictorAgent* running a neural network on a GPU which evaluates a board and predicts the associated reward. Co-locating *BoardAggregator*s and their corresponding *PredictorAgent*s on the same physical node is thus desirable to avoid network overheads from transferring board states. These pairings also require anti-affinity for load balancing and to avoid interference [190]. With ephemeral resources, this composed policy can be specified in 5 lines of code (fig. 4.6a): we apply a load-balancing policy (Table 4.2) to the *PredictorAgent* and a co-location policy to the *BoardAggregator* and *PredictorAgent*.

We ran 10k iterations of AlphaZero on a 32-node cluster (128 GPUs total). We compare three setups: (a) co-location with hard-coded placement, (b) co-location with ephemeral resources, and (c) a baseline policy with no co-location. Figure 4.6b plots the CDF for board exploration time. Co-location is important for performance, outperforming no-colocation by 15.4% in median latency and 20% in P95 latency. Additionally, co-location with ephemeral resources adds insignificant overheads of <1%, while requiring less developer effort: the application code (Figure 4.6a) does not need to match *PredictorAgent-BoardAggregator* pairs to specific nodes.

Distributed Training

For a distributed training job, worker placement is critical to performance, as co-locating workers reduces the cost of model synchronization at each step. Gandiva [190] is a scheduler for deep learning jobs that aims to optimize training job performance. It composes a higher level *load-balancing* policy and a lower-level *co-location* policy to evenly spread jobs across machines while reducing intra-job communication overhead. To demonstrate ESCHER's flexibility, we augment Gandiva's [190] worker co-location and migration policy with Gang Scheduling to support distributed training jobs, and integrate the policy into Tune [96], an open source distributed training library built on

Ray [118], which we will refer to as *EscherTune*. We modified the Trial abstraction in Tune to be wrapped in a ghost task that ensures gang scheduling and applied co-location on tasks belonging to the same Trial. EscherTune triggers a migration whenever it detects sufficient available resources to place all workers of a job on the same node. To execute a worker migration, EscherTune checkpoints the current job using application-specific checkpoint functionality and destroys all current workers. Then, EscherTune assigns ephemeral resources to the new target node, and relaunches all worker tasks of the training job without modifying their ephemeral resource requests.

We compare EscherTune with Tune's open-source policy on a cluster of 12 GPUs. We launch 5 short-running training jobs (*short-jobs*), each requiring 1 GPU, followed by 1 long-running training job requiring 4 GPUs (*long-job*). Each training job is training a ResNet-101 model on CIFAR-10 with a batch-size of 64 images per device.

Initially, the *short-jobs* are load-balanced across the cluster, while the 4 workers of the *long-job* are spread across the cluster depending on GPU availability. This is a sub-optimal placement, so EscherTune migrates the *long-job* to colocate its tasks as soon as resources become available from a *short-job* completion, resulting in 36.3% higher throughput (Figure 4.6c). Meanwhile, Tune uses a static placement, so the *long-job*'s throughput remains the same. Furthermore, EscherTune's implementation consists of only 50 lines of Python, with no changes to Tune or the Ray scheduler.

Microbenchmarks

Overhead of application-level policies

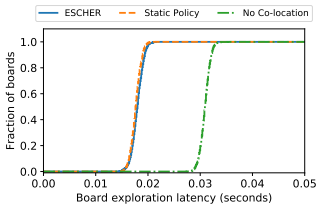
ESCHER scheduling policies can be implemented either in the application space for evolvability or in the framework for performance. We evaluate the trade-offs involved in this choice by comparing three distinct designs of gang scheduling, all with ephemeral resources on Ray.

AppSpace uses ghost tasks to atomically reserve resources (Section 4.4). While this policy is simple to integrate, the lack of coordination between applications can lead to deadlock, which must be resolved through timeouts. *LibSpace* avoids this by using a *shared library*: a shared service in the cluster that serializes gang scheduling requests across applications. *LibSpace* thus avoids live lock entirely but requires deploying a separate shared service. Finally, *FrameSpace* modifies the Ray scheduler to expose a gang scheduling API. Internally, a centralized service within Ray directly reserves and creates ephemeral resources. Since it has direct access to the resource table, *FrameSpace* avoids using ghost tasks, reducing overheads from worker allocation and task dispatch.

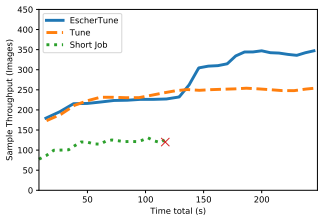
Figure 4.7 compares the request latency of these designs on a 32-node cluster with 256 CPUs. While the mean latency of *AppSpace* and *LibSpace* is similar, *AppSpace* has higher variance and a longer tail because it uses timeouts to break deadlocks. *LibSpace* incurs overhead from serializing requests at a separate service, resulting in a higher minimum latency. On average, *FrameSpace* is nearly $2\times$ faster than *AppSpace* and *LibSpace* because it directly reserves resources instead of using ghost tasks. However, we note that for long-running tasks such as model training and batch processing workloads, the absolute scheduling latency is still a tiny fraction (<1s) compared to the runtime of the workloads (multiple hours). Moreover, implementing *FrameSpace* is a significant

```
            class
                PredictorAgent():
              def
                  __init__(id):
                #
                    Create
                    co-location
                    resource.

                    set_resource(

                      name=id,

                      capacity=1)
                ...
def main():
  # Create load-balancing
      resources
  for node in cluster:
      set_resource("load_bal", 1,
      node)
  for i in range(0, num_agents):
    p = PredictorAgent(resources
        = {"GPU": 1, "load_bal":
        1}).launch(id=i)
    # The predictor creates a
        resource with label i
    # This resource is used by
        the BoardAgg to
        co-locate.
    b = BoardAggregator(resources
        = {i: 1}).launch(p)
```

(a)



(b)



(c)

Figure 4.6: AlphaZero and distributed training on ESCHER. **(a)** Implementing AlphaZero policy with ESCHER, composing co-location with load-balancing. **(b)** A CDF of AlphaZero board exploration latency, and **(c)** Throughput comparison of a distributed training workload with a mix of short-running and long-running jobs. EscherTune is an augmentation of the hyperparameter search framework Tune [96], using ESCHER to dynamically re-schedule jobs as others complete. The red X indicates the completion of a short job.

effort, requiring a deep understanding of the Ray scheduler and modifying 1624 lines of Ray code. To compare, *LibSpace* and *AppSpace* are implemented in 261 and 78 lines of *application-level* code, respectively.

Overheads of Ephemeral Resources

We evaluate the time to create resources and propagate their availability throughout the cluster. Since the `set_resource` call is asynchronous, we verify that the resources have been created and are available for use by launching no-op tasks that request these newly created resources. Figure 4.8a compares the mean latency of creating an equal number of resources on each node in a 50-node Ray cluster. We show that even when creating 1000 ephemeral resources, we can maintain

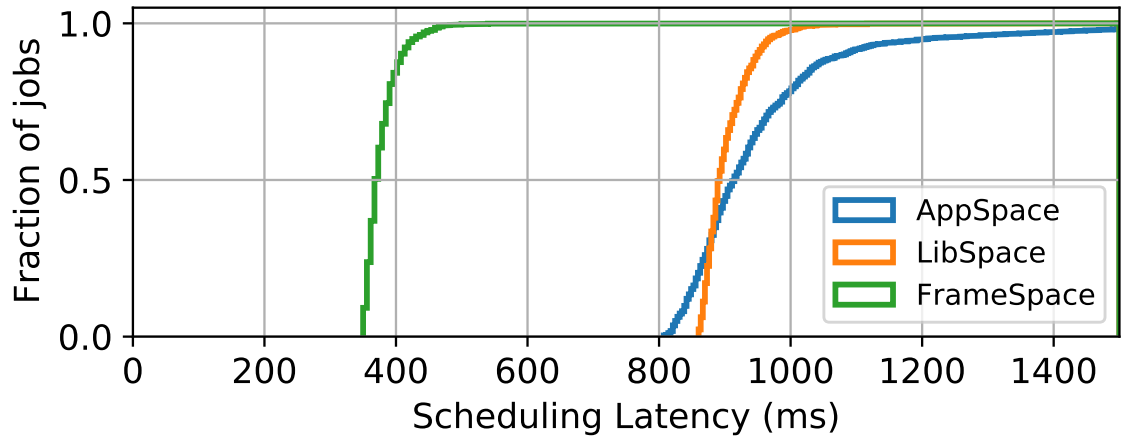
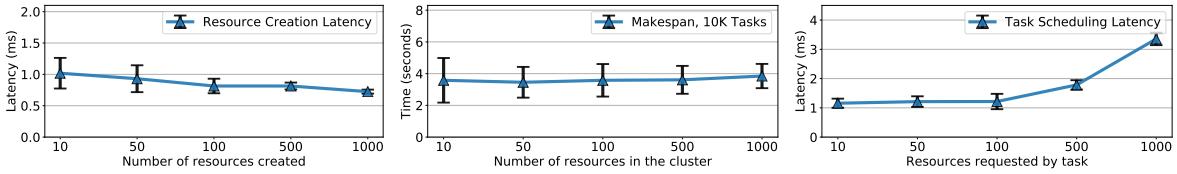


Figure 4.7: Request latency for gang scheduling implemented in the application space, with (*LibSpace*) and without (*AppSpace*) coordination, versus the framework space (*FrameSpace*). *FrameSpace* is 1624 lines of code (LoC), *LibSpace* with 261 LoC and *AppSpace* with 78 LoC.



(a) **(b)** **(c)**

Figure 4.8: ESCHER microbenchmarks. **(a)** Mean per-resource creation latency in Ray. Creating ephemeral resources in ESCHER is a low-cost operation that scales linearly with the number of resources created. **(b)** Scheduling latency overheads from presence of ephemeral resources. Makespan of a 10000 task workload remains unaffected by the count of ephemeral resources in the cluster. **(c)** Effect of task resource requirements on scheduling latency in an environment with 10000 resources.

1ms latency per request. As more resources are created, the cost of resource creation is amortized and the per-resource creation cost decreases to 0.72ms. In general, the overhead of creating or deleting an ephemeral resource should be roughly equivalent to that of a key-value store request.

Ephemeral resources and scheduling latency. The creation of ephemeral resources may add burden to the scheduler, as it must consider a greater number of attributes during resource matching. Therefore, we analyze the effect of resource creation on task scheduling latency. We create an equal number of resources across 50 Ray nodes in a cluster using the `set_resource` API. We then evaluate two cases based on the resource requirements of the tasks involved.

First, in Figure 4.8b, we launch 10,000 tasks, none of which require any ephemeral resources to be scheduled. As the tasks do not have any specific resource requirements, the scheduler execution time and workload makespan are not affected by the number of ephemeral resources present.

Second, when tasks do request ephemeral resources, the core scheduler must match the task's requirements to a set of candidate nodes. To evaluate the overheads introduced by this matching,

we setup a 50 node Ray cluster and create 1000 unique ephemeral resources evenly spread across nodes. Figure 8c highlights the scalability of the scheduler as the number of ephemeral resources requested by a task grows. The the task scheduling latency grows only from 1.1ms to 1.2ms when requesting 1 vs. 100 ephemeral resources, respectively. We note that all policies described in this work require only a few ephemeral resources to express.

4.7 Discussion

Why use ESCHER? For common scheduling policies, ESCHER's primary benefit compared to a monolithic cluster manager is not performance. Rather, it's the ease to specify and implement new policies without requiring any changes to the cluster scheduler. This unlocks developing new applications with sophisticated scheduling constraints that are not yet supported by the underlying scheduler. One example is the *composition* of affinity and gang scheduling policies used in the distributed training example in Section 4.6, which is not supported by Ray's native scheduling primitives. Another example is DAG-based scheduling, which is offered natively by Ray but not Kubernetes. DAG-based scheduling can be implemented by leveraging the task signaling primitive (Section 4.4). Effectively, ESCHER expands the set of policies a monolithic cluster manager can support.

Two-level schedulers [154, 65] achieve the same goal by exporting scheduling control directly to applications, but in doing so they also require applications to implement the entire scheduler themselves (Section 4.3). ESCHER on the other hand requires minimal changes to application code: it required adding only two lines of code to MapReduce, five lines to AlphaZero and fifty lines to EscherTune, each of which had widely varying policy requirements. For policy composition especially, this ease of development is due in part to the use of ESLs.

Limitations. A key goal of ephemeral resources is to provide a simple and narrow API that is easy to implement by most cluster managers. As a result, ESCHER eschews abstractions that would require complex implementations such as transaction support [154] (which would allow to trivially implement gang scheduling) or utilization-based scheduling, such as load balancing (Section 4.4). While the application can still implement these policies using ghost tasks, these implementations have inherently a higher overhead. Of course, if applications require higher scheduling performance, we can eventually implement these policies in the cluster manager. Even in this case, ESCHER remains valuable as it can bridge the gap by enabling the applications to implement these policies before the cluster manager does.

Another limitation of ESCHER is that it doesn't expose the resource availability to the application, which means that the only way for an application to learn that there are not enough resources available is by submitting a task which hangs. As a result, the application might have to explicitly kill the tasks that hang, adding to the complexity. We do not expose resource availability is because it would not fully solve the problem: there is still a race condition when two tasks on different machines simultaneously request more than the available resources (e.g., a single GPU is available and two tasks simultaneously request one GPU each). This problem is exacerbated by the fact that ephemeral resources can be dynamically created, modified, or destroyed. One solution to this

problem is providing transaction semantics, which, as mentioned above, ESCHER eschews due to its complex implementation. An avenue for future work would be to alleviate the challenges of handling such a dynamic environment, e.g., by using lazy execution or extending the API to allow constraints on ephemeral resources.

4.8 Related Work

Monolithic schedulers. Monolithic cluster schedulers aim to implement both the scheduling policy and mechanism for a distributed application. Some provide a single generic scheduling discipline, such as fair sharing [70, 50], gang scheduling [59], or delay scheduling [193]. These schedulers provide little control to the application beyond the ability to set some configuration knobs, such as the time to wait before scheduling a task on a node that doesn't store its inputs [193].

Other monolithic schedulers aim for generality and provide APIs to allow applications to express scheduling constraints. Examples are YARN [179], Condor's ClassAds [97], and Kubernetes [22]. Their monolithic design makes it hard to add support for new policies and their compositions. For instance, adding support for gang scheduling to Kubernetes requires deep structural and API changes, and was eventually implemented as a standalone system [109]. Often the API they provide is complex as well, since it must be expressive enough to capture complex constraints such as composition. For instance, the specification of ClassAds [139] is 35 pages [165]. ESCHER achieves both *simplicity* and *evolvability* by decoupling policy specification and the resource-matching mechanism.

Like ESCHER, DCM [167] also aims to maximize extensibility of framework schedulers by using a declarative model for applications to specify their desired policy behavior as SQL queries. In doing so, DCM deploys a custom scheduler and optimizer running with Kubernetes. While this is well suited for policies which optimize for global objectives, expressing application-level constraints requires users to create and maintain a table in cluster state database, which can be challenging in a distributed environment. ESCHER's emphasis lies on supporting application-level scheduling goals, allowing it to easily handle task-task dependencies (Primitive P2 in section 4.4). Moreover, ESCHER reuses existing virtual resource implementations, thus requiring no additional services or schedulers to be deployed in the cluster framework.

Rayon [32] is a space-time reservation admission system, allowing applications to reserve a *skyline* of resource capacity, $c(t)$, as a function of time. ESCHER can implement a discrete version of this by having a ghost task evaluate $c(t)$ and update ephemeral resources to match $c(t)$ at any instant.

Two-level schedulers. Rather than trying to implement application level policies, some cluster management frameworks are designed explicitly to give all resource management and scheduling control to the application. Many of these frameworks employ a two-level hierarchy [179, 65], where the first level manages only resource isolation between applications, while the second level exposes physical resources to applications. These applications are then responsible for building their own scheduler. Omega [154] follows a similar separation by providing transaction semantics on a shared cluster state for distributed schedulers. While this approach grants maximum flexibility

to applications, it adds significant complexity to application code since the application must now handle both scheduling policy and the mechanisms to ensure resource coordination between tasks. Some popular frameworks, such as Spark [194] and Flink [23] obviate the need for distributed coordination by designating a special node (e.g., master) to spawn all tasks. However, they too have monolithic designs that are not evolvable. In contrast, ESCHER focuses on providing a generic scheduling framework where the application only focuses on the scheduling policy. Indeed, ESCHER can be used in tandem with two-level schedulers by launching an ESCHER scheduler to manage resources allocated by the top-level scheduler.

Label-based and declarative scheduling. [22, 179, 154, 139, 181] provide mechanisms to annotate nodes with resource types and use these labels (e.g., "GPU:Nvidia:V100") for placement constraints. In some cases, these labels do not have an associated capacity (e.g., string key-value pairs), rendering infeasible implementation of policies with *quantitative* conditions. In other cases, quantitative labels are static. TetriSched [175] operates on labelled resources by allowing declarative resource constraint specification and composition. Wrasse [138] uses the bins and balls abstraction along with user-defined utilization functions to come up with a specification language. However, neither of these provide support for *dynamic* scheduling policies, e.g., making inter-task constraints hard to implement in a single shot. The expressivity of declarative schedulers is restricted to information known *a priori* (i.e., static label information). Circular inter-task dependencies ([119]) are fundamentally impossible to implement without a dynamic mechanism to unroll the dependency (Section 4.2). We note, however, that declarative schedulers ([175, 167]) are synergistic with ESCHER. Ephemeral resources can be used as an intermediate representation (IR) for their frontend API (e.g., SQL [167] and STRL [175]).

Some existing schedulers provide the ability to configure non-physical resources, e.g., the *extended resources* API in Kubernetes [12]. The original purpose of this mechanism is for the cluster operator to add accounting for custom resources (e.g., accelerators). Meanwhile, generic application-level scheduling policies like affinity and load balancing are still implemented in the Kubernetes core. In contrast, ESCHER obviates the need to implement these policies in the core scheduler, deferring it to the application level via the mechanism of ephemeral resources. In fact, the ESCHER implementation on Kubernetes repurposes extended resources to implement *all* scheduling policies and simplifies the Kubernetes scheduler to only resource matching. Finally, the ability to dynamically update ephemeral resources at runtime enables expressing previously inexpressible, e.g., inter-task “happens-before” relationships and iterative task graphs.

4.9 Conclusion

Designing cluster scheduling frameworks that can evolve with changing requirements of applications presents a three way trade-off between evolvability, application simplicity and performance. With *ephemeral resources*, ESCHER marks a new point in this design space which makes cluster frameworks evolvable by allowing applications to implement a wide range of scheduling policies without taking on the complexities of scheduler design and implementation. This gain in evolvability is valuable for many applications, for whom functionality in the short-term (while the

policy is integrated into the scheduler) may be more important than performance. Further, ESCHER Scheduling Libraries (ESLs) reduce application level complexity by encapsulating policy implementations in a portable layer. Finally, we implement ESCHER on Kubernetes and Ray to illustrate the generality of ESCHER and achieve performance comparable to hard-coded schedulers on a variety of machine learning and data processing workloads.

4.10 Acknowledgements

We thank the SoCC reviewers and our shepherd, Lin Wang, for their invaluable feedback. This research is partly supported by NSF (CCF-1730628) and gifts from Amazon, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

Chapter 5

Conclusion

5.1 Conclusion

1. Talk about how efficiency helped reduce supply-demand gap
2. Future Work - Other ideas, user experience for resouce allocation
3. Future Work - Explain that availability is also contributes to supply - SkyPilot solves availability

Bibliography

[1] A Comprehensive List of Hyperparameter Optimization & Tuning Solutions. https://medium.com/@mikkokotila/a-comprehensive-list-of-hyperparameter-optimization-tuning-solutions-88e067f19d9. 2018.

[2] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. "Fair Allocation of Multiple Resource Types". In: *USENIX NSDI*. 2011.

[3] Martin Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *USENIX OSDI*. 2016.

[4] Achieving Compliant Data Residency and Security with Azure. In.

[5] AI and Compute. *https://openai.com/blog/ai-and-compute/*. 2018.

[6] *Amazon Compute Service Level Agreement*. 2022. URL: %7Bhttps://aws.amazon.com/compute/sla/%7D (visited on 09/21/2021).

[7] Ganesh Ananthanarayanan et al. "Real-time Video Analytics – the killer app for edge computing". In: *IEEE Computer* (2017).

[8] Thomas Anthony et al. *Policy Gradient Search: Online Planning and Expert Iteration without Search Trees*. 2019. eprint: arXiv:1904.03646.

[9] Peter Auer. "Using confidence bounds for exploitation-exploration trade-offs". In: *Journal of Machine Learning Research* 3.Nov (2002), pp. 397–422.

[10] Kubeflow authors. *Kubeflow Homepage*. 2022. URL: https://www.kubeflow.org/ (visited on 10/07/2022).

[11] Kubernetes Authors. *Concepts: Kubernetes Scheduler*. 2022. URL: https://kubernetes.io/docs/concepts/scheduling/ (visited on 10/07/2022).

[12] Kubernetes authors. *Managing Compute Resources for Containers - Kubernetes*. 2022. URL: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/ (visited on 10/07/2022).

[13] AWS Outposts. *https://aws.amazon.com/outposts/*.

[14] *Azure Percept*. https://azure.microsoft.com/en-us/services/azure-percept/.

[15] Azure Stack Edge. *https://azure.microsoft.com/en-us/services/databox/edge/*.

[16] Eden Belouadah and Adrian Popescu. “IL2M: Class Incremental Learning With Dual Memory”. In: *IEEE ICCV*. 2019.

[17] Jon CR Bennett and Hui Zhang. “WF/sup 2/Q: worst-case fair weighted fair queueing”. In: *Proceedings of IEEE INFOCOM’96. Conference on Computer Communications*. Vol. 1. IEEE. 1996, pp. 120–128.

[18] James Bergstra and Yoshua Bengio. “Random Search for Hyper-Parameter Optimization”. In: *J. Mach. Learn. Res.* 13 (2012), pp. 281–305.

[19] Keith Bonawitz et al. “Towards Federated Learning at Scale: System Design”. In: *SysML*. 2019.

[20] Sébastien Bubeck and Nicolo Cesa-Bianchi. “Regret analysis of stochastic and nonstochastic multi-armed bandit problems”. In: *arXiv preprint arXiv:1204.5721* (2012).

[21] Sébastien Bubeck et al. “X-armed Bandits”. In: *arXiv preprint arXiv:1001.4475* (2010).

[22] Brendan Burns et al. “Borg, Omega, and Kubernetes”. In: *ACM Queue* 14 (2016), pp. 70–93. URL: http://queue.acm.org/detail.cfm?id=2898444.

[23] Paris Carbone et al. “Apache flink: Stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[24] Abhishek Chandra et al. “Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors”. In: *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation - Volume 4*. OSDI’00. San Diego, California: USENIX Association, 2000.

[25] Shuang Chen, Christina Delimitrou, and José F Martínez. “Parties: Qos-aware resource partitioning for multiple interactive services”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 107–120.

[26] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, Matthai Philipose. “VideoEdge: Processing Camera Streams using Hierarchical Clusters”. In: *ACM/IEEE SEC*. 2018.

[27] Dah-Ming Chiu and Raj Jain. “Analysis of the increase and decrease algorithms for congestion avoidance in computer networks”. In: *Computer Networks and ISDN systems* 17.1 (1989), pp. 1–14.

[28] CLIFFORD, M. J., PERRONS, R. K., ALI, S. H.,ANDGRICE, T. A. “Extracting Innovations: Mining, Energy, and Technological Changein the Digital Age”. In: *CRC Press*. 2018.

[29] cnn-benchmarks. *https://github.com/jcjohnson/cnn-benchmarks#resnet-101*.

[30] Andrea Coraddu et al. “Machine learning approaches for improving condition-based maintenance of naval propulsion plants”. In: *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment* 230.1 (2016), pp. 136–153.

[31] Daniel Crankshaw et al. “Clipper: A low-latency online prediction serving system”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 613–627.

[32] Carlo Curino et al. “Reservation-Based Scheduling: If You’re Late Don’t Blame Us!” In: SOCC ’14. Seattle, WA, USA: Association for Computing Machinery, 2014, pp. 1–14. ISBN: 9781450332521. DOI: 10.1145/2670979.2670981. URL: https://doi.org/10.1145/2670979.2670981.

[33] D Maltoni, V Lomonaco. “Continuous learning in single-incremental-task scenarios”. In: *Neural Networks*. 2019.

[34] D. Kang, J. Emmons, F. Abuzaid, P. Bailis and M. Zaharia. “NoScope: Optimizing Neural Network Queries over Video at Scale”. In: *VLDB*. 2017.

[35] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[36] Jeffrey Dean et al. “Large Scale Distributed Deep Networks”. In: *NeurIPS*. 2012.

[37] Christina Delimitrou and Christos Kozyrakis. “Paragon: QoS-aware scheduling for heterogeneous datacenters”. In: *ACM SIGPLAN Notices* 48.4 (2013), pp. 77–88.

[38] Christina Delimitrou and Christos Kozyrakis. “Quasar: resource-efficient and QoS-aware cluster management”. In: *ACM SIGPLAN Notices* 49.4 (2014), pp. 127–144.

[39] Alan Demers, Srinivasan Keshav, and Scott Shenker. “Analysis and simulation of a fair queueing algorithm”. In: *ACM SIGCOMM Computer Communication Review* 19.4 (1989), pp. 1–12.

[40] Danny Dolev et al. “No justified complaints: On fair sharing of multiple resources”. In: *proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. 2012, pp. 68–75.

[41] Kuntai Du et al. “Server-Driven Video Streaming for Deep Learning Inference”. In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 2020, pp. 557–570.

[42] Edge Computing at Chick-fil-A. “https://medium.com/@cfatechblog/edge-computing-at-chick-fil-a-7d67242675e2”. In: 2019.

[43] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. "Exokernel: An Operating System Architecture for Application-level Resource Management". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 251–266. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224076.

[44] Andrew D. Ferguson et al. "Jockey: Guaranteed Job Latency in Data Parallel Clusters". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: Association for Computing Machinery, 2012, pp. 99–112. ISBN: 9781450312233. DOI: 10.1145/2168836.2168847. URL: https://doi.org/10.1145/2168836.2168847.

[45] Andrew D. Ferguson et al. "Jockey: Guaranteed Job Latency in Data Parallel Clusters". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: Association for Computing Machinery, 2012, pp. 99–112. ISBN: 9781450312233. DOI: 10.1145/2168836.2168847. URL: https://doi.org/10.1145/2168836.2168847.

[46] Kelwin Fernandes, Pedro Vinagre, and Paulo Cortez. "A proactive intelligent decision support system for predicting the popularity of online news". In: *Portuguese Conference on Artificial Intelligence*. 2015.

[47] Brad Fitzpatrick. "Distributed caching with memcached". In: *Linux journal* 124 (2004).

[48] Yu Gan et al. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18. ISBN: 9781450362405. DOI: 10.1145/3297858.3304013. URL: https://doi.org/10.1145/3297858.3304013.

[49] Ganesh Ananthanarayanan, Victor Bahl, Yuanchao Shu, Franz Loewenherz, Daniel Lai, Darcy Akers, Peiwei Cao, Fan Xia, Jiangbo Zhang, Ashley Song. "Traffic Video Analytics – Case Study Report". In: 2019.

[50] Ali Ghodsi et al. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types." In: *Nsdi*. Vol. 11. 2011. 2011, pp. 24–24.

[51] Ali Ghodsi et al. "Multi-resource Fair Queueing for Packet Processing". In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 2012, pp. 1–12.

[52] GI Parisi, R Kemker, JL Part, C Kanan, S Wermter. "Continual lifelong learning with neural networks: A review". In: *Neural Networks*. 2019.

[53] Daniel Golovin et al. "Google Vizier: A Service for Black-Box Optimization". In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '17. Halifax, NS, Canada: Association for Computing Machinery, 2017, pp. 1487–1495. ISBN: 9781450348874. DOI: 10.1145/3097983.3098043. URL: https://doi.org/10.1145/3097983.3098043.

[54] *Google AI Blog: Custom On-Device ML Models with Learn2Compress*. https://ai.googleblog.com/2018/05/custom-on-device-ml-models.html. (Accessed on 03/09/2021).

[55] Alkis Gotovos. "Active learning for level set estimation". MA thesis. Eidgenössische Technische Hochschule Zürich, Department of Computer Science, 2013.

[56] Robert Grandl et al. "Altruistic scheduling in multi-resource clusters". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 65–80.

[57] Robert Grandl et al. "Multi-resource packing for cluster schedulers". In: *ACM SIGCOMM*. 2014.

[58] Jean-Bastien Grill, Michal Valko, and Rémi Munos. "Black-box optimization of noisy functions with unknown smoothness". In: *Advances in Neural Information Processing Systems*. 2015, pp. 667–675.

[59] William D Gropp et al. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press, 1999.

[60] Juncheng Gu et al. "Tiresias: A GPU Cluster Manager for Distributed Deep Learning". In: *USENIX NSDI*. 2019.

[61] Avital Gutman and Noam Nisan. "Fair allocation without trade". In: *arXiv preprint arXiv:1204.4286* (2012).

[62] *Hadoop Fair Scheduler*. 2022. URL: https://hadoop.apache.org/.

[63] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodík, Matthai Philipose, Victor Bahl, Michael Freedman. "Live Video Analytics at Scale with Approximation and Delay-Tolerance". In: *USENIX NSDI*. 2017.

[64] Benjamin Hindman et al. "Mesos: A platform for fine-grained resource sharing in the data center." In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

[65] Benjamin Hindman et al. "Mesos: A platform for fine-grained resource sharing in the data center." In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

[66] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. "Distilling the Knowledge in a Neural Network". In: *NeurIPS Deep Learning and Representation Learning Workshop*. 2015.

[67] Kevin Hsieh et al. "Focus: Querying Large Video Datasets with Low Latency and Low Cost". In: *USENIX OSDI*. 2018.

[68] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "Sequential Model-Based Optimization for General Algorithm Configuration". In: *Learning and Intelligent Optimization*. 2011.

[69] Michael Isard et al. “Quincy: fair scheduling for distributed computing clusters”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 261–276.

[70] Michael Isard et al. “Quincy: fair scheduling for distributed computing clusters”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 261–276.

[71] Raj Jain, Dah-Ming Chiu, and W. Hawe. “A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems”. In: *CoRR* cs.NI/9809099 (1998). URL: https://arxiv.org/abs/cs/9809099.

[72] Junchen Jiang, Vyas Sekar, and Hui Zhang. “Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive”. In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. 2012, pp. 97–108.

[73] Joseph Redmon, Ali Farhadi. “YOLO9000: Better, Faster, Stronger”. In: *CVPR*. 2017.

[74] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodík, Siddhartha Sen, Ion Stoica. “Chameleon: Scalable Adaptation of Video Analytics”. In: *ACM SIGCOMM*. 2018.

[75] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyengar, Pillai Padmanabhan, Mahadev Satyanarayanan. “Towards Scalable Edge-Native Applications”. In: *ACM/IEEE Symposium on Edge Computing*. 2019.

[76] Sangeetha Abdu Jyothi et al. “Morpheus: Towards Automated SLOs for Enterprise Clusters”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 117–134. ISBN: 978-1-931971-33-1.

[77] K He, X Zhang, S Ren, J Sun. “Deep Residual Learning for Image Recognition”. In: *CVPR*. 2016.

[78] M. Frans Kaashoek et al. “Application Performance and Flexibility on Exokernel Systems”. In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. SOSP ’97. Saint Malo, France: ACM, 1997, pp. 52–65. ISBN: 0-89791-916-5. DOI: 10.1145/268998.266644. URL: http://doi.acm.org/10.1145/268998.266644.

[79] Vasiliki Kalavri et al. “Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 783–798.

[80] Faria Kalim et al. “Henge: Intent-Driven Multi-Tenant Stream Processing”. In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC ’18. Carlsbad, CA, USA: Association for Computing Machinery, 2018, pp. 249–262. ISBN: 9781450360111. DOI: 10.1145/3267809.3267832. URL: https://doi.org/10.1145/3267809.3267832.

[81] Kirthevasan Kandasamy et al. "Online Learning Demands in Max-min Fairness". In: *arXiv preprint arXiv:2012.08648* (2020).

[82] Mamoru Kaneko and Kenjiro Nakamura. "The Nash social welfare function". In: *Econometrica: Journal of the Econometric Society* (1979), pp. 423–435.

[83] Frank P Kelly, Aman K Maulloo, and David KH Tan. "Rate control for communication networks: shadow prices, proportional fairness and stability". In: *Journal of the Operational Research society* 49.3 (1998), pp. 237–252.

[84] Mehrdad Khani et al. "Real-Time Video Inference on Edge Devices via Adaptive Model Streaming". In: *arXiv preprint arXiv:2006.06628* (2020).

[85] Jaehong Kim et al. "Neural-Enhanced Live Streaming: Improving Live Video Ingest via Online Learning". In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 107–125. ISBN: 9781450379557. DOI: 10.1145/3387514.3405856. URL: https://doi.org/10.1145/3387514.3405856.

[86] Konstantin Shmelkov, Cordelia Schmid, Karteek Alahari. "Incremental Learning of Object Detectors Without Catastrophic Forgetting". In: *ICCV*. 2017.

[87] *Kubernetes API health endpoints | Kubernetes*. https://kubernetes.io/docs/reference/using-api/health-checks/. 2022.

[88] Kevin Lai et al. "Tycoon: An implementation of a distributed, market-based resource allocation system". In: *Multiagent and Grid Systems* 1.3 (2005), pp. 169–182.

[89] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[90] Jeongtae Lee et al. "Lifelong Learning with Dynamically Expandable Networks". In: *ICLR*. 2018.

[91] Ang Li et al. "A Generalized Framework for Population Based Training". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '19. Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 1791–1799. ISBN: 9781450362016. DOI: 10.1145/3292500.3330649. URL: https://doi.org/10.1145/3292500.3330649.

[92] Lisha Li et al. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization". In: *J. Mach. Learn. Res.* 18.1 (Jan. 2017), pp. 6765–6816. ISSN: 1532-4435.

[93] Lisha Li et al. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization". In: *J. Mach. Learn. Res.* 18 (2017), 185:1–185:52.

[94] Yuanqi Li et al. "Reducto: On-Camera Filtering for Resource-Efficient Real-Time Video Analytics". In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 2020, pp. 359–376.

[95] Richard Liaw et al. "HyperSched: Dynamic Resource Reallocation for Model Development on a Deadline". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, pp. 61–73. ISBN: 9781450369732. DOI: 10.1145/3357223.3362719. URL: https://doi.org/10.1145/3357223.3362719.

[96] Richard Liaw et al. "Tune: A research platform for distributed model selection and training". In: *arXiv preprint arXiv:1807.05118* (2018).

[97] Michel J Litzkow, Miron Livny, and Matt W Mutka. *Condor-a hunter of idle workstations*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 1987.

[98] Wei Liu et al. "SSD: Single shot multibox detector". In: *European Conference on Computer Vision*. Springer. 2016, pp. 21–37.

[99] Yucheng Low et al. "Distributed GraphLab: A Framework for Machine Learning in the Cloud". In: *PVLDB* 5.8 (2012), pp. 716–727.

[100] Yan Lu et al. "Collaborative learning between cloud and end devices: an empirical study on location prediction". In: *ACM/IEEE SEC*. 2019.

[101] M McCloskey, NJ Cohen. "Catastrophic interference in connectionist networks: The sequential learning problem". In: *Psychology of learning and motivation*. 1989.

[102] M Sandler, A Howard, Menglong Zhu, Andrey Zhmoginov, Liang-Chieh Chen. "Mobilenetv2: Inverted residuals and linear bottlenecks". In: *CVPR*. 2018.

[103] Klaus Ma. *Coscheduling in Kubernetes · Issue #61012 · kubernetes/kubernetes*. 2018. URL: https://github.com/kubernetes/kubernetes/issues/61012 (visited on 10/07/2022).

[104] Klaus Ma. *Coscheduling. by k82cn · Pull Request #639 · kubernetes/enhancements*. 2018. URL: https://github.com/kubernetes/enhancements/pull/639 (visited on 10/07/2022).

[105] Michael J. Magazine and Maw-Sheng Chern. "A Note on Approximation Schemes for Multidimensional Knapsack Problems". In: *Math. Oper. Res.* 9.2 (1984).

[106] Kshiteej Mahajan et al. "Themis: Fair and Efficient GPU Cluster Scheduling". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 289–304. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/mahajan.

[107] Kshiteej Mahajan et al. "Themis: Fair and Efficient GPU Cluster Scheduling for Machine Learning Workloads". In: *USENIX NSDI*. 2020.

[108] Spyros Makridakis and Michele Hibon. "ARMA models and the Box–Jenkins methodology". In: *Journal of forecasting* 16.3 (1997), pp. 147–163.

[109] mali11. *Schedule a group of pods all at once (aka coscheduling, sometimes known as gang scheduling)*. 2015. URL: https://github.com/kubernetes/kubernetes/issues/16845 (visited on 10/07/2022).

[110] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. "The Cityscapes Dataset for Semantic Urban Scene Understanding". In: *CVPR*. 2016.

[111] Measuring Fixed Broadband - Eighth Report, FEDERAL COMMUNICATIONS COMMISSION OFFICE OF ENGINEERING AND TECHNOLOGY. "https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-eighth-report". In: 2018.

[112] Microsoft-Rocket-Video-Analytics-Platform. *https://github.com/microsoft/Microsoft-Rocket-Video-Analytics-Platform*.

[113] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, Quoc V. Le. "MnasNet: Platform-Aware Neural Architecture Search for Mobile". In: *CVPR*. 2019.

[114] Mingxing Tan, Quoc V. Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *ICML*. 2019.

[115] Ujval Misra et al. "RubberBand: Cloud-Based Hyperparameter Tuning". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 327–342. ISBN: 9781450383349. URL: https://doi.org/10.1145/3447786.3456245.

[116] Jeonghoon Mo and Jean Walrand. "Fair end-to-end window-based congestion control". In: *IEEE/ACM Transactions on networking* 8.5 (2000), pp. 556–567.

[117] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 561–577. ISBN: 9781931971478.

[118] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018.

[119] Derek G. Murray et al. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 439–455. ISBN: 9781450323888. DOI: 10.1145/2517349.2522738. URL: https://doi.org/10.1145/2517349.2522738.

[120] *MxNet: a flexible and efficient library for deep learning*. https://mxnet.apache.org/.

[121] Arun Nair et al. “Massively parallel methods for deep reinforcement learning”. In: *arXiv preprint arXiv:1507.04296* (2015).

[122] Raghunath Othayoth Nambiar and Meikel Poess. “The Making of TPC-DS”. In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB ’06. Seoul, Korea: VLDB Endowment, 2006, pp. 1049–1058.

[123] Deepak Narayanan et al. “Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 481–498. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak.

[124] Vikram Nathan et al. “End-to-end transport for video QoE fairness”. In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 408–423.

[125] Hiep Chi Nguyen et al. “AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service”. In: *International Conference on Automation and Computing*. 2013.

[126] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. “ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design”. In: *ECCV*. 2018.

[127] *Nvidia Multi-Process Service*. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. (Accessed on 09/16/2020).

[128] OPENSIGNAL. Mobile Network Experience Report. “https://www.opensignal.com/reports/2019/01/usa/n network-experience”. In: 2019.

[129] Kay Ousterhout et al. “Sparrow: Distributed, Low Latency Scheduling”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 69–84. ISBN: 9781450323888. DOI: 10.1145/2517349.2522716. URL: https://doi.org/10.1145/2517349.2522716.

[130] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[131] R. H. Patterson et al. “Informed Prefetching and Caching”. In: *SIGOPS Oper. Syst. Rev.* 29.5 (1995), pp. 79–95. ISSN: 0163-5980. DOI: 10.1145/224057.224064. URL: https://doi.org/10.1145/224057.224064.

[132] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, Jan Kautz. “Pruning Convolutional Neural Networks for Resource Efficient Inference”. In: *ICLR*. 2017.

[133] Pei Sun and Henrik Kretzschmar and Xerxes Dotiwalla and Aurelien Chouard and Vijaysai Patnaik and Paul Tsui and James Guo and Yin Zhou and Yuning Chai and Benjamin Caine and Vijay Vasudevan and Wei Han and Jiquan Ngiam and Hang Zhao and Aleksei Timofeev and Scott Ettinger and Maxim Krivokon and Amy Gao and Aditya Joshi and Yu Zhang and Jonathon Shlens and Zhifeng Chen and Dragomir Anguelov. *Scalability in Perception for Autonomous Driving: Waymo Open Dataset*. 2019.

[134] Yanghua Peng et al. "Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190517. URL: https://doi.org/10.1145/3190508.3190517.

[135] Yanghua Peng et al. "Optimus: an efficient dynamic resource scheduler for deep learning clusters". In: *ACM EuroSys*. 2018.

[136] Devin Petersohn. "Scaling Interactive Data Science Transparently with Modin". MA thesis. Berkeley, CA: UC Berkeley, 2018.

[137] Haoran Qiu et al. "FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 805–825. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/qiu.

[138] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. "Generalized Resource Allocation for the Cloud". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: Association for Computing Machinery, 2012. ISBN: 9781450317610. DOI: 10.1145/2391229.2391244. URL: https://doi.org/10.1145/2391229.2391244.

[139] R. Raman, M. Livny, and M. Solomon. "Matchmaking: distributed resource management for high throughput computing". In: *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No.98TB100244)*. 1998, pp. 140–146. DOI: 10.1109/HPDC.1998.709966.

[140] Jeff Rasley et al. "HyperDrive: exploring hyperparameters with POP scheduling". In: *ACM/IFIP/USENIX Middleware*. 2017.

[141] Ravi Teja Mullapudi, Steven Chen, Keyi Zhang, Deva Ramanan, Kayvon Fatahalian. "Online Model Distillation for Efficient Video Inference". In: *ICCV*. 2019.

[142] Suman V. Ravuri and Oriol Vinyals. "Classification Accuracy Score for Conditional Generative Models". In: 2019.

[143] *Ray Dashboard — Ray v1.7.0*. https://docs.ray.io/en/latest/ray-dashboard.html. 2022.

[144] Ali Sharif Razavian et al. "CNN Features off-the-shelf: an Astounding Baseline for Recognition". In: *IEEE CVPR Workshop*. 2014.

[145] Joseph Redmon and Ali Farhadi. *YOLOv3: An Incremental Improvement*. 2018. arXiv: 1804.02767 [cs.CV].

[146] *Reducing Edge Compute Cost for Live Video Analytics*. https://techcommunity.microsoft.com/t5/internet-of-things/live-video-analytics-with-microsoft-rocket-for-reducing-edge/ba-p/1522305. (Accessed on 03/09/2021).

[147] Charles Reiss et al. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: Association for Computing Machinery, 2012. ISBN: 9781450317610. DOI: 10.1145/2391229.2391236. URL: https://doi.org/10.1145/2391229.2391236.

[148] Residential landline and fixed broadband services. "https://www.ofcom.org.uk/__data/assets/pdf_file/0015 broadband.pdf". In: 2019.

[149] RM French. "Catastrophic forgetting in connectionist networks". In: *Trends in cognitive sciences*. 1999.

[150] Herbert Robbins. "Some aspects of the sequential design of experiments". In: *Bulletin of the American Mathematical Society* 58.5 (1952).

[151] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler L. Hayes, and Christopher Kanan. "Measuring Catastrophic Forgetting in Neural Networks". In: *AAAI*. 2018.

[152] Krzysztof Rzadca et al. "Autopilot: workload autoscaling at Google". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[153] Hadoop Fair Scheduler. "https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html". In.

[154] Malte Schwarzkopf et al. "Omega: Flexible, Scalable Schedulers for Large Compute Clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 351–364. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465386. URL: http://doi.acm.org/10.1145/2465351.2465386.

[155] *scipy.optimize.nnls — SciPy v1.5.2 Reference Guide*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.nnls.html. (Accessed on 09/17/2020).

[156] Shadi Noghabi, Landon Cox, Sharad Agarwal, Ganesh Ananthanarayanan. "The Emerging Landscape of Edge-Computing". In: *ACM SIGMOBILE GetMobile*. 2020.

[157] Haichen Shen et al. "Fast Video Classification via Adaptive Cascading of Deep Models". In: *CVPR*. 2017.

[158] Shivangi Srivastava, Maxim Berman, Matthew B. Blaschko, Devis Tuia. "Adaptive Compression-based Lifelong Learning". In: *BMVC*. 2019.

[159] Konstantin Shvachko et al. “The hadoop distributed file system”. In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee. 2010, pp. 1–10.

[160] Si Young Jang, Yoonhyung Lee, Byoungheon Shin, Dongman Lee, Dionisio Vendrell Jacinto. “Application-aware IoT Camera Virtualization for Video Analytics Edge Computing”. In: *ACM/IEEE SEC*. 2018.

[161] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529 (2016), pp. 484–503. URL: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[162] David Silver et al. “Mastering the game of go without human knowledge”. In: *Nature* 550.7676 (2017), p. 354.

[163] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. “Practical bayesian optimization of machine learning algorithms”. In: *Advances in neural information processing systems* 25 (2012).

[164] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *NIPS*. 2012.

[165] Marvin Solomon. *The ClassAd Language Reference Manual Version 2.4*. https://research.cs.wisc.edu/htcondor/classad/refman.pdf. 2004.

[166] Song Han, Huizi Mao, William J. Dally. “Accelerating Very Deep Convolutional Networks for Classification and Detection”. In: *ICLR*. 2017.

[167] Lalith Suresh et al. “Building Scalable and Flexible Cluster Managers Using Declarative Programming”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 827–844. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/suresh.

[168] Sweden Data Collection & Processing. In.

[169] Kevin Swersky et al. “Scalable Bayesian Optimization Using Deep Neural Networks”. In: *ICML*. 2015.

[170] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, Christoph H. Lampert. “iCaRL: Incremental Classifier and Representation Learning”. In: *CVPR*. 2017.

[171] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. 5th. USA: Prentice Hall Press, 2010. ISBN: 0132126958.

[172] Gil Tene. *giltene/wrk2: A constant throughput, correct latency recording variant of wrk*. https://github.com/giltene/wrk2. (Accessed on 04/19/2022).

[173] The Future of Computing is Distributed. *https://www.datanami.com/2020/02/26/the-future-of-computing-is-distributed/*. 2020.

[174] *torchvision.models — PyTorch 1.6.0 documentation*. https://pytorch.org/docs/stable/torchvision/models.html. (Accessed on 09/16/2020).

[175] Alexey Tumanov et al. "TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters". In: *Proc. of the 11th European Conference on Computer Systems*. EuroSys '16. London, UK: ACM, 2016.

[176] Yan Kyaw Tun et al. "Wireless network slicing: Generalized kelly mechanism-based resource allocation". In: *IEEE Journal on Selected Areas in Communications* 37.8 (2019), pp. 1794–1807.

[177] *Twitter Streaming API*. 2022. URL: https://developer.twitter.com (visited on 05/31/2020).

[178] Hal R Varian. "Equity, envy, and efficiency". In: (1973).

[179] Vinod Kumar Vavilapalli et al. "Apache hadoop yarn: Yet another resource negotiator". In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 5.

[180] Shivaram Venkataraman et al. "Ernest: Efficient performance prediction for large-scale advanced analytics". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 2016, pp. 363–378.

[181] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[182] Huiyu Wang et al. "Elastic: Improving cnns with dynamic scaling policies". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2258–2267.

[183] Max Welling. "Kernel ridge regression". In: *Max Welling's classnotes in machine learning* (2013), pp. 1–3.

[184] John Wilkes. "Utility Functions, Prices, and Negotiation". In: *Market-Oriented Grid and Utility Computing*. John Wiley and Sons, Ltd, 2009. Chap. 4, pp. 67–88. ISBN: 9780470455432.

[185] Yue Wu et al. "Large Scale Incremental Learning". In: *IEEE CVPR*. 2019.

[186] Xi Yin, Xiang Yu, Kihyuk Sohn, Xiaoming Liu and Manmohan Chandraker. "Feature Transfer Learning for Face Recognition with Under-Represented Data". In: *IEEE CVPR*. 2019.

[187] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". In: *IEEE PAMI*. 2016.

[188] Wencong Xiao et al. "Gandiva: Introspective Cluster Scheduling for Deep Learning". In: *USENIX OSDI*. 2018.

[189] Wencong Xiao et al. "Gandiva: Introspective cluster scheduling for deep learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 595–610.

[190] Wencong Xiao et al. "Gandiva: Introspective cluster scheduling for deep learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 595–610.

[191] Reynold Xin. *Project Hydrogen: Unifying State-of-the-art AI and Big Data in Apache Spark*. San Francisco, CA, 2018. URL: https://databricks.com/session/databricks-keynote-2.

[192] Z. Li and D. Hoiem. "Learning without forgetting". In: *ECCV*. 2016.

[193] Matei Zaharia et al. "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling". In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 265–278.

[194] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, p. 10.

[195] Matei Zaharia et al. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[196] Seyed Majid Zahedi, Qiuyun Llull, and Benjamin C Lee. "Amdahl's law in the datacenter era: A market for fair processor allocation". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 1–14.

[197] Haoyu Zhang et al. "SLAQ: quality-driven scheduling for distributed machine learning". In: *SoCC*. 2017.

[198] Xiangyu Zhang et al. "Shufflenet: An extremely efficient convolutional neural network for mobile devices". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 6848–6856.

[199] Yanqi Zhang et al. "Sinan: ML-based and QoS-aware resource management for cloud microservices". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 167–181.

[200] Yuchen Zhang, John Duchi, and Martin Wainwright. "Divide and conquer kernel ridge regression". In: *Conference on learning theory*. PMLR. 2013, pp. 592–617.

[201] Hang Zhu et al. "RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1225–1240. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/zhu.

[202] Ziwei Liu, Zhongqi Miao, Xiaohang Zhan, Jiayun Wang, Boqing Gong, Stella X. Yu. "Large-Scale Long-Tailed Recognition in an Open World". In: *CVPR*. 2019.