BOA Constrictor: Squeezing Performance out of GPUs in the Cloud via Budget-Optimal Allocation

Zhouzi Li
zhouzil@andrew.cmu.edu
Carnegie Mellon University
Computer Science Department
United States

Cindy Zhu
cindyz@andrew.cmu.edu
Carnegie Mellon University
Computer Science Department
United States

Arpan Mukhopadhyay
arpan.mukhopadhyay@warwick.ac.uk
University of Warwick
Computer Science Department
England

Mor Harchol-Balter
harchol@cs.cmu.edu
Carnegie Mellon University
Computer Science Department
United States

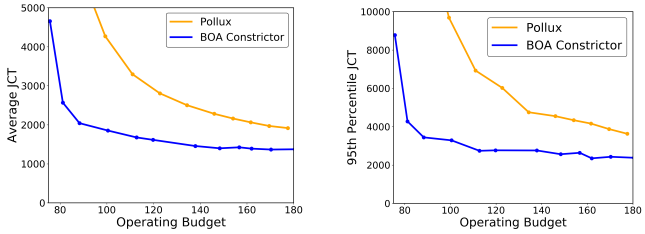
Benjamin Berg
ben@cs.unc.edu
UNC Chapel Hill
Computer Science Department
United States

arXiv:2602.01404v2 [cs.DC] 26 Jul 2026

Abstract

The past decade has seen a dramatic increase in demand for GPUs to train Machine Learning (ML) models. Because it is prohibitively expensive for most organizations to build and maintain a large GPU cluster, organizations instead rent GPUs from cloud providers. A *cloud customer* must decide (i) how many GPUs to rent at every moment in time to process a stream of training jobs and (ii) how to allocate the rented GPUs among the currently active jobs. While allocating more GPUs to a single training job helps the job complete more quickly, the customer pays for each GPU-hour they use. Because training jobs often receive a diminishing marginal benefit from running on additional GPUs, allocating too many GPUs to a single job can dramatically increase the cost the customer pays. This gives rise to a *cost-performance tradeoff* when training models in the cloud.

To balance the cost-performance tradeoff, we develop BOA Constrictor, a new scheduler for ML training jobs that uses a Budget-Optimal Allocation (BOA) policy to squeeze the most performance out of a cloud-based GPU cluster given a fixed budget. While prior approaches focus on fixed-sized clusters and heuristic approaches for balancing cost and performance, we formalize the problem as a budget-constrained scheduling problem. Given a monetary budget that the customer is willing to spend on GPUs, we derive the BOA policy that minimizes the average job completion time (JCT) of a stream of arriving jobs. Our BOA policy can be computed efficiently for any budget level, providing users with the optimal tradeoff between cost and performance. For a given budget level, BOA Constrictor can reduce average JCT by 1.6× in small-scale implementation experiments and by 2× in detailed, large-scale simulations compared to state-of-the-art schedulers. BOA Constrictor also reduces the budget needed to achieve a given average JCT by up to 2×.



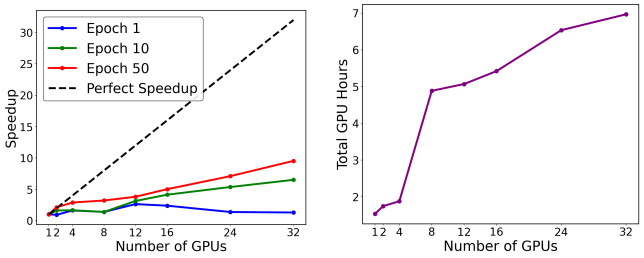
(a) Average JCT vs. budget. **(b)** P95 JCT vs. budget.

Figure 1. BOA Constrictor balances the cost-performance tradeoff for training in the cloud. While existing policies such as Pollux [27] do not explicitly balance the cost-performance tradeoff, BOA Constrictor provides up to a 2× improvement in average JCT and up to a 3× improvement in P95 JCT for a given budget by deriving a budget-optimal allocation policy.

1 Introduction

The explosion in Machine Learning (ML) over the past ten years has led to a dramatic increase in demand for GPUs to train ML models [17]. Because it is prohibitively expensive for many organizations to build and maintain a large GPU cluster, hyperscale cloud providers (Microsoft Azure, Amazon AWS, Google Cloud) have seen explosive growth in demand for renting cloud-based GPUs [2]. This raises the crucial question of how cloud customers can efficiently train ML models using cloud-based GPU clusters [16, 22, 23, 27].

In this cloud computing paradigm, the customer is responsible for devising a *GPU rental policy* that decides how many GPUs to rent at every moment in time [28]. Here, the *customer* may be an individual developer or researcher, or an organization such as a company or research lab that rents a pool of GPUs to serve a stream of ML training jobs that are submitted over time. The customer is charged on a pay-for-what-you-use basis according to a set fee per GPU-Hour of usage. The customer may increase or decrease their rental



(a) Speedup functions **(b)** Training cost

Figure 2. Performance of a Cifar10 training job on different numbers of GPUs. Changes to job hyperparameters over the course of training change the job's speedup function. Because the speedup functions are sublinear, training costs increase when the job runs on a larger number of GPUs.

demands dynamically over time to both suit the performance needs of their workloads and control costs.

The cost-performance tradeoff. Renting additional GPUs may decrease *job completion times* (JCTs), but can also increase the time-average cost paid by the customer. Because cloud-based GPU instances can be rented or returned on a timescale of 1 - 2 minutes, customers can scale their clusters up or down in response to changes in the current workload. This raises the question of what rental policy a customer should employ to scale their cluster in real time and balance a tradeoff between achieving good performance and limiting the costs paid to the cloud provider. We note several key *challenges* that complicate the cost-performance tradeoff:

C.1: Training jobs receive sublinear speedup. Modern ML training jobs can be parallelized across multiple GPUs to complete training more quickly. However, the benefits of parallelism are tempered by a combination of sequential bottlenecks, synchronization overheads, and statistical inefficiency (see Section 2.1). As a result, training jobs generally receive a diminishing marginal benefit from running on additional GPUs. This effect is commonly measured via a job's *speedup function*, $s(k)$, which describes how fast a job makes progress when training a model on k GPUs. Due to the aforementioned limits to parallelism, a training job's speedup function is generally sublinear and concave.

Sublinear speedup functions have a major impact on the cost-performance tradeoff. If a job receives a linear speedup, it can run on k GPUs and complete in $\frac{1}{k}$th the time, meaning that its total GPU usage remains constant. In this case, the cost of training is independent of the rental policy, and every job should run on as many GPUs as possible. Because speedup functions are sublinear in practice, a job run on k GPUs will complete in $\frac{1}{s(k)}$ as much time, leading to a factor $\frac{k}{s(k)}$ *increase* in the number of GPU-hours required to complete the job (Figure 2b). Hence, the choice of rental policy can have a massive impact on both cost and performance.

C.2: Training jobs and training workloads change over time. Balancing the cost-performance tradeoff for cloud-based ML training is particularly challenging because the problem is dynamic along several dimensions. First, a job's speedup function may change over the course of training (see Figure 2a) as the optimal choice of job hyperparameters changes (see Section 2.1) [16, 27, 34]. Second, a customer does not deal with a single job, but a stream of training jobs that are submitted over time. Each job may have a different speedup function that changes as the job runs. The composition of the training workload may also shift over time. Third, job arrivals can exhibit significant bursts. We confirm that real-world traces exhibit both bursts and composition changes over time in Section 2.2. As a result, rental policies designed to optimize the execution of a single job (see Section 2.3) do not generalize to handle a stream of jobs.

C.3: Existing cluster schedulers fall short. Broadly speaking, balancing the cost-performance tradeoff involves two principal decisions that must be made in real time:

1. How many GPUs to rent from the cloud provider at every moment in time?
2. How to divide the rented GPUs among the training jobs currently in the system?

For example, perhaps the cluster size should grow to avoid queueing when a burst of arrivals occurs, or perhaps GPUs should be shifted away from less-parallelizable jobs to more-parallelizable jobs over time to control costs.

As detailed in Section 2.3, *existing schedulers address at most one of the above questions*, with the majority of existing cluster schedulers focusing on fixed-size clusters. Prior approaches have the added drawback that they generally make use of scheduling heuristics aimed at maximizing *cluster-wide efficiency* (see Section 2.3). We show that efficiency-based heuristics are provably suboptimal and perform poorly in practice compared to theoretically-grounded policies.

Our Solution: BOA Constrictor. This paper presents *BOA Constrictor*, a scheduling framework for cost-efficient cloud-based training of ML models. To control costs, the customer provides BOA Constrictor with an operating *budget*, the long-run average number of GPU-hours they wish to expend per hour. Given a stream of training jobs, at every moment of time, BOA Constrictor determines (i) the overall cluster size, and (ii) how to allocate the rented GPUs to each job in the system. Based on the customer's budget and the overall workload composition, BOA Constrictor minimizes the *average job completion time* (average JCT) across jobs. For example, the customer might aim to spend $10,000 per month on training, which is equivalent to maintaining an average cluster size of 40 GPUs. Over the course of a day, BOA Constrictor will adjust the cluster size and job allocations. Note that both of these rescaling actions come with overheads, so BOA Constrictor must limit the frequency of these changes.

The core of BOA Constrictor is our *Budget-Optimal Allocation* policy (BOA), which minimizes average JCT subject to a budget constraint under a very general system model. Our BOA policy exposes the entire Pareto frontier between cost and performance, allowing the customer to optimally balance the cost-performance tradeoff according to their own preferences (see Figure 1).

Contributions

The contributions of this paper are as follows:

- We formally model the *GPU rental problem* in which a cloud service customer has a budget on the long-run average number of GPU-hours consumed per hour. The customer aims to minimize the average JCT (or weighted average JCT) across a stream of jobs while adhering to the budget. Our stochastic model (Section 3) makes very mild distributional assumptions, allowing us to accurately model real-world systems.
- We solve the GPU rental problem in Section 4. Our results handle both homogeneous (Section 4.1) and heterogeneous (Section 4.2) clusters. We refer to our optimal policy as the Budget Optimal Allocation (BOA) policy.
- We implement BOA Constrictor, a cluster scheduler that leverages the BOA policy, using AdaptDL [1] (Section 5). Using both our implementation and detailed ML training simulators, we evaluate BOA Constrictor on production traces (Section 6), comparing it to several state-of-the-art cluster schedulers. While existing systems mainly target fixed-size clusters, we also develop autoscaling variants of each system to compare against BOA Constrictor.
- For a given budget, BOA Constrictor improves average JCT by up to a factor of ~1.75× in implementation (Figure 1). Additionally, for a given average JCT, BOA Constrictor reduces the budget needed to achieve this average JCT by up to a factor of ~2.2×. BOA Constrictor balances the cost-performance tradeoff by using our theoretical results to minimize queueing time and limit job preemption/rescaling instead of relying on the intuitive scheduling heuristics used by state-of-the-art systems.

2 Background

This section describes the current state-of-the-art in training ML models in the cloud. We describe how modern ML training jobs can be dynamically configured to leverage multiple GPUs (Section 2.1), why training workloads require dynamic scheduling and autoscaling policies (Section 2.2), and how state-of-the-art approaches for scheduling training jobs fall short both in theory and in practice (Section 2.3), revealing the need for a theoretically-grounded policy that optimizes the cost-performance tradeoff as a primary concern.

2.1 Distributed Training in the cloud

To accelerate the ML training process, training jobs are typically distributed across multiple GPUs in parallel. However, making efficient use of a set of GPUs to accelerate training requires careful configuration of the training job.

Exploiting parallelism in distributed training. A single job can exploit parallelism in two central ways. First, *data parallelism* involves replicating a full model across many GPUs and dividing batches of training data across these replicas to process a higher number of samples in parallel per unit time. Second, *model parallelism* involves spreading parts of a single model instance across multiple GPUs, either by partitioning different layers of the model onto separate GPUs (pipeline parallelism) or splitting a single tensor of model weights across multiple GPUs (tensor parallelism).

This paper mainly considers data parallel jobs, but we consider other modes of parallelism in Section 6.1. This choice eases our comparison with the prior work. We track a job's training progress via its *accuracy*, which we measure as the training loss with respect to a customer-defined loss function. We consider a job to be complete when it reaches a desired level of accuracy. As described in [27], data-parallel training jobs experience two main overheads from parallelization. First, these jobs periodically synchronize with a parameter server to compute gradients. This limits the job's *throughput*, the number of training examples processed per second. Second, computing gradients on higher numbers of GPUs in parallel requires larger batches of data to be processed between gradient updates, reducing the training *efficiency*, defined as the accuracy gained per training sample [27]. Combined, these effects give rise to the sublinear speedup functions measured for the data parallel jobs in Figure 2a. Similar overheads produce sublinear speedup functions when different modes of parallelism are used [34].

Hyperparameter selection. Training jobs expose a vast array of *hyperparameters* that can be tuned to control the performance characteristics of the training process. These hyperparameters range from training algorithm parameters like learning rate and batch size [27] to higher-level decisions such as the mode of parallelism [34] or the type of model being trained. The choice of hyperparameters can significantly affect the speed of training [19], and finding the optimal configuration is known to be difficult. Nonetheless, techniques such as Bayesian optimization and multi-armed bandit theory have been used to explore the hyperparameter space and efficiently find good hyperparameter configurations.

Recent works have noted that the choice of a degree of parallelism affects the optimal choices of other hyperparameters (e.g., optimal batch size depends on the degree of parallelism [27]). Several papers suggest intertwining the hyperparameter search process with the scheduling algorithm that chooses the degree of parallelism for each job

[8, 16, 22, 27, 34]. We observe that this work tends to leverage highly sophisticated hyperparameter search techniques in combination with overly simplistic scheduling heuristics. This suggests that there is significant room for improvement solely by optimizing the scheduling aspect of these systems.

Our Approach. Our approach is to decouple scheduling from the hyperparameter search. We assume that for every job, a hyperparameter search has already discovered a good configuration to use with each possible number of GPUs. That is, the speedup functions considered in our model (Section 3) represent the speedup a job receives given the best known set of hyperparameters for that degree of parallelism. We show in Section 6 that our approach is general enough to improve performance across several systems that make different choices at lower levels in the system. That is, our work is *complementary* to prior work on optimizing distributed training and hyperparameter selection. We show that BOA Constrictor is beneficial when different hyperparameter tuning strategies and different modes of parallelism are used.

2.2 Workloads

To understand why modern training workloads can benefit so greatly from improved scheduling on flexibly-sized clusters, we note that modern workloads like [16, 27] exhibit notable variability along several dimensions:

Job running times are highly variable. Given a fixed number of GPUs, the time required to complete a training job can vary by more than an order of magnitude depending on the model being trained and the level of accuracy required. Improved scheduling policies are known to reduce queueing time when job running times are highly variable [14].

Job arrivals are bursty. Jobs experience several bursts of arrivals over time. Leveraging cluster autoscaling can reduce the effects of variability in the arrival process [10].

A job's speedup function changes over time. A job's speedup function depends on the job's throughput and efficiency. As noted in [27], job efficiency tends to be lower at the beginning of training. Hence, job speedup functions tend to shift upwards over the course of training.

For a full description of the workloads used in our evaluation, see Section 6.1. These observations about training workloads suggest that systems could benefit from more complex, theoretically grounded allocation policies.

2.3 Prior Work: Current Schedulers Fall Short

Prior theoretical work. Prior theoretical work has separately considered the problem of scheduling parallelizable jobs [3, 4, 6, 7] and autoscaling [9, 10, 26], but we are not aware of any work which considers both problems *simultaneously* to balance the cost-performance tradeoff. By developing a stochastic model of this problem and solving for the BOA policy in Sections 3 and 4, we provide a novel theoretical contribution on how real-world systems can benefit by optimally balancing the cost-performance tradeoff.

Prior systems fall broadly into one of three categories.

Approach 1: Reservation-based systems. Reservation-based schedulers like Ray [23] and Tiresias [13] and others [15, 33] ask the customer to specify the GPU requirements of each of their jobs. For these systems, the goal is to provide the resources demanded for each job as quickly as possible. To this end, Ray uses very simple heuristics related to data locality and load balancing, but does not explicitly aim to minimize the average JCT across jobs. Tiresias recommends using the Gittins index as a scheduling heuristic, ignoring the fact the Gittins index is suboptimal when scheduling multiserver jobs outside of very heavy load [30]. Ray allows autoscaling, while Tiresias does not, but all reservation-based systems inherently defer the cost-performance tradeoff to the customer and provide no decision support in how to set GPU requirements. Moreover, relying on customer-generated reservations also prevents the system from changing job allocations to respond to changes in a job or the overall workload.

Approach 2: Policy-based allocation with a fixed cluster size. A wide variety of recently proposed systems try to improve performance by a combination of hyperparameter tuning and dynamic GPU allocation [12, 15, 16, 18, 20, 21, 24, 25, 27, 34, 35]. While approaches differ in exactly how job configurations are tuned, which heuristics are used, and which performance metrics are optimized (e.g., makespan, JCT, or fairness), these systems are similar in their high-level goals of choosing GPU allocations to optimize overall performance across a stream of training jobs. All prior work only considers a fixed-size cluster (no autoscaling) (except for [27], which will be discussed in Approach 3).

This paper considers minimizing the *average job completion time* (JCT) across a stream of training jobs that arrive to the system over time. Among the above works that optimize for JCT, we note that Pollux [27], Rubick [34], and Sia [16] represent the state of the art. [1] We therefore compare our work against these three systems in Section 6 to show that our budget-optimal allocation policy for a cloud-based cluster greatly outperforms these fixed-cluster systems.

Approach 3: Autoscaling for a single job. Work on autoscaling to balance the cost-performance tradeoff has focused on the setting of running a single job [27, 32]. We note that these approaches do not generalize to handling a stream of arriving jobs. Specifically, in Section 6 we develop augmented versions of Pollux, Sia, and Rubick based on the target efficiency autoscaling mechanism proposed for a single job in [27]. These autoscaling variants significantly underperform our solution, BOA Constrictor.

[1] Lucid [15] can outperform Pollux, but only at very high loads.

3 Our Model

To minimize notation, we largely limit our description of the model to the case of a homogeneous cluster; we introduce additional notation when we get to the case of a heterogeneous cluster in Section 4.2. We model the GPU rental problem from the perspective of a *customer* who submits a stream of ML training jobs to be run in the cloud. Unlike prior work, which makes highly restrictive assumptions such as a Poisson arrival process, our model makes very few assumptions about the workload and system. This allows our results to apply across a wide range of real-world scenarios.

Training jobs. We abstractly view each training job as being associated with some *inherent work* that quantifies the statistical progress required for the model to reach a desired level of accuracy. The customer has M classes of training jobs, each corresponding to a different combination of model and training data source. We use random variable X_i to denote the inherent work of a type-i job. The inherent work of different jobs is *not necessarily independent.*

Each training job has an associated speedup function, $s(k)$, that specifies the *rate* at which inherent work is processed when the job runs on k GPUs. As noted in Section 2.1, we take $s(k)$ to be the *best* rate attainable on k GPUs after optimizing over the job's hyperparameters (e.g., batch size, learning rate, parallelism strategy) and the placement of those GPUs across physical nodes. This decouples the allocation decision from hyperparameter selection and placement, which are handled by orthogonal mechanisms in our system (see Section 5).

A job's speedup function may vary as training progresses. Following [27], we capture this by dividing each type-i job into ℓ_i statistical epochs, where epoch $j \in \{1, \ldots, \ell_i\}$ has random inherent work X_{ij}, with finite mean $\mathbb{E}[X_{ij}]$, and speedup function $s_{ij}(k)$. Let $X_i = \sum_{j=1}^{\ell_i} X_{ij}$. A type-$i$ job run on k GPUs during epoch j completes in time $X_{ij}/s_{ij}(k)$.

For each class i, we let $\mathcal{K}_i$ denote the minimum number of GPUs on which a class-i job can run; the speedup function $s_{ij}(k)$ is defined on $k \in [\mathcal{K}_i, \infty)$. This captures the resource requirements of jobs that cannot fit onto a small number of GPUs (e.g., $\mathcal{K}_i = 8$ for Large Language Model fine-tuning). For ease of analysis, we allow the number of GPUs allocated to a job to be fractional; prior work [11] shows that fractional allocations can be rounded to integral allocations without significant loss of efficiency.[2]

Arrival process and load. Our system processes a stream of jobs that arrive at a long-run average rate of λ jobs/sec. Type-i jobs arrive at rate λ_i jobs/sec. We define the *system load* contributed by type-i jobs to be $\rho_i = \lambda_i \mathbb{E}[X_i]$. Intuitively, ρ_i represents the long-run average rate at which type-i work is submitted to the system. To describe the load contributed by individual job epochs, we define $\rho_{ij} = \lambda_i \mathbb{E}[X_{ij}]$ to be the load of the jth epoch of the ith job type, so $\rho_i = \sum_j \rho_{ij}$.

Rescaling. Changing a job's GPU allocation incurs *rescaling overhead.* This is the time required to checkpoint the job, set up the runtime environment, download container images and training data, and resume the job on the new GPUs. We denote the rescaling time of a class-i job by a random variable R_i with mean $\mathbb{E}[R_i] = r_i$.

Cost-performance tradeoff. Our goal is to design *allocation policies* that trade off cost and performance by renting GPUs and assigning them to jobs over time.

In terms of *performance*, this paper optimizes Job Completion Time (JCT), the time from when a job arrives until it completes. Let T_i denote the JCT of a type-i job and $\mathbb{E}[T_i]$ its average. We typically optimize $\mathbb{E}[T]$, the overall *average JCT* across a stream of arriving jobs, where $\mathbb{E}[T]$ is given by

$$\mathbb{E}[T] = \sum_{i=1}^{M} \frac{\lambda_i}{\lambda} \mathbb{E}[T_i].$$

Our results easily extend to any weighted average JCT, where each class is assigned a weight to capture fairness or the relative importance of different jobs. Although we do not explicitly optimize for tail latency, the evaluation in Section 6 shows that BOA Constrictor also improves P95-JCT.

We measure *cost* in monetary units (dollars). In the case of homogeneous GPUs, we assume that each GPU is rented for a constant cost, so the dollar cost is equivalent to the number of GPU-hours used. We denote the time-average cost by $\bar{K}$. In the heterogeneous setting of Section 4.2, different GPU types have different dollar-per-hour rates.

The Cost-performance tradeoff: Given a customer's time-average operating budget of b GPUs, the goal is to find a policy π that minimizes the average JCT subject to $\bar{K} \leq b$.

Modeling assumptions. We have made three incredibly general modeling assumptions, which we summarize here.

First, for the stochastic quantities in the system—interarrival times, inherent work, and rescaling times—we assume that the time averages for sequences of each quantity converge to some finite mean with probability 1. We assume these averages are known to the customer. We make no other distributional assumptions. Notably, we allow correlations between the inherent work of different jobs. Furthermore, we allow correlations between interarrival times. This allows us to model bursty, non-stationary workloads.

Second, for each type i and epoch j, the speedup function $s_{ij}(k)$ should fulfill the following properties: (1) $s_{ij}(k)$ is defined and continuous on $k \in [\mathcal{K}_i, +\infty)$; (2) *Monotonicity*: $s_{ij}(k_1) \leq s_{ij}(k_2)$ for any $\mathcal{K}_i \leq k_1 < k_2$; (3) *Concavity*: $s_{ij}(k_1)/k_1 \geq s_{ij}(k_2)/k_2$ for any $\mathcal{K}_i \leq k_1 < k_2$. In practice, naively measured speedup functions like those in Figure 2a may violate the latter two assumptions. However, we can always remedy this by considering the monotonic concave envelope of the speedup function. Prior work [11] has shown

[2]The BOA policy we derive can be easily adapted to handle physically partitioned GPUs [29] (e.g., NVIDIA Multi-instance GPUs), but evaluating BOA Constrictor under these conditions is outside the scope of this paper.

that one can achieve performance equivalent to this concave envelope without incurring additional rescaling by alternating between allocations on the envelope over time. We follow this approach to handle non-concave speedup functions in our implementation (see Algorithm 1).

Third, we assume the budget is high enough to keep up with the stream of training jobs.

4 Budget-Optimal Allocation

The goal of BOA is to allocate GPUs to minimize (weighted) average job completion time, $\mathbb{E}[T]$, while keeping the time-average cost under a budget, b. Before we formally define BOA, we first develop some intuition for BOA in the simplest case of homogeneous clusters without rescaling overheads.

There are many intuitions that are popular in the scheduling theory literature, such as "finish short-running jobs first" [6], or "prioritize the less parallelizable jobs" [5]. There are also other heuristics used in prior systems (discussed in detail in Section 6), such as "maximize total speedup across jobs"[3] [16, 27], or "autoscale based on cluster efficiency" [27]. It turns out that all these intuitions are suboptimal for our problem setting. Instead, we prove that the *optimal policy* has two somewhat counter-intuitive structural properties:

1. **No job queues under the optimal policy (Lemma A.3)**. Forcing jobs to queue may reduce the *instantaneous* number of GPUs used, but it does not reduce the long-run average GPU usage. Hence, queueing is suboptimal because it worsens JCTs without improving budget usage.
2. **The optimal policy is a *fixed-width* policy in that the number of GPUs allocated to a job depends only on its current speedup function. Specifically, type-i jobs in the same epoch j are allocated the same number of GPUs, k_{ij} (Lemma A.4).** Because speedup functions are concave, a job makes the best use of a fixed number of GPU-hours by running on a constant number of GPUs rather than changing allocations. This phenomenon also applies across jobs of the same type/epoch.

Both properties are proved in Appendix A. Together they pin down the behavior of the optimal policy: every job is allocated some number of GPUs immediately upon arrival, and its allocation changes only when the job enters its next epoch. New GPUs are requested or released only in response to meeting the requirements of each class/epoch. We refer to this type of policy as a *fixed-width policy*.

Definition 4.1 (Fixed-width policy)**.** A *fixed-width* policy is parameterized by a set of constants $\{k_{ij}\}$: It assigns k_{ij} GPUs to every type-i job for the entire duration of epoch j, without queueing.

Definition 4.2 (BOA policy)**.** The BOA policy is defined to be the best fixed-width policy. That is, for any budget b, the BOA policy is the fixed-width policy that chooses the values of k_{ij} to minimize (weighted) average JCT subject to b.

We now show that the BOA policy is optimal in both homogeneous (Section 4.1) and heterogeneous (Section 4.2) clusters when there are no rescaling overheads. We adapt the BOA policy to handle rescaling overheads in Section 4.3.

4.1 BOA in Homogeneous Clusters

Theorem 4.3 (Optimality of BOA)**.** *In the homogeneous setting without rescaling overheads, the budget-optimal allocation (BOA) policy is optimal and can be computed by solving the optimization problem:*

$$\begin{aligned} \underset{\{k_{ij}\}}{\text{minimize}} \quad & \text{average JCT} = \mathbb{E}[T] = \frac{1}{\lambda}\sum_i \lambda_i \sum_j \left(\frac{\mathbb{E}[X_{ij}]}{s_{ij}(k_{ij})}\right) \\ \text{subject to} \quad & \text{operating budget} = \sum_i \lambda_i \sum_j k_{ij}\left(\frac{\mathbb{E}[X_{ij}]}{s_{ij}(k_{ij})}\right) \le b, \\ & k_{ij} \ge \mathcal{K}_i. \end{aligned}$$

This is a convex optimization problem.

Proof. In Appendix A, we provide a slightly more formal statement of Theorem 4.3 (Theorem A.1) and then prove this theorem via a series of lemmas.

Roughly, we prove that the optimal policy is a fixed-width policy and compute its optimal "widths", k_{ij}. Under a fixed-width policy with parameters $\{k_{ij}\}$, every type-i job in epoch j runs on k_{ij} GPUs. This epoch completes in expected time $\mathbb{E}[T_{ij}] = \mathbb{E}[X_{ij}]/s_{ij}(k_{ij})$. Summing over classes and epochs and weighting by arrival rates yields the average JCT:

$$\mathbb{E}[T] \quad = \quad \frac{1}{\lambda}\sum_i \lambda_i \sum_j \mathbb{E}[T_{ij}] = \frac{1}{\lambda}\sum_i \lambda_i \sum_j \frac{\mathbb{E}[X_{ij}]}{s_{ij}(k_{ij})} \quad (1)$$

Similarly, every type-i job in epoch j holds k_{ij} GPUs for time $\mathbb{E}[T_{ij}]$, consuming $k_{ij} \cdot \mathbb{E}[T_{ij}]$ GPU-hours. This yields the time-average operating budget:

$$\sum_i \lambda_i \sum_j k_{ij}\mathbb{E}[T_{ij}] = \sum_i \lambda_i \sum_j k_{ij}\frac{\mathbb{E}[X_{ij}]}{s_{ij}(k_{ij})} \quad (2)$$

Choosing $\{k_{ij}\}$ to minimize the average JCT subject to the operating budget being at most b then reduces to a constrained convex optimization problem which can be solved efficiently via numerical methods. □

4.2 BOA in Heterogeneous Clusters

We now extend BOA to the heterogeneous setting. Here we consider H different clusters, each composed of a different type of GPU. A customer can rent GPUs from any cluster, but GPUs from each cluster have distinct performance profiles and rental costs. Following [16], we make the mild assumption that each job runs on a single type of GPUs at a time.

For each class i, epoch j, and GPU type h, the speedup $s_{ij}^{(h)}(k)$ denotes the rate at which inherent work is processed

[3] This is similar to the concept of *water-filling* [31].

on k type-h GPUs; $c^{(h)}$ denotes the per-hour rental cost of one type-h GPU; $\mathcal{K}_i^{(h)}$ denotes the minimum number of type-h GPUs required to run a class-i job.

Theorem 4.4 generalizes Theorem 4.3 to heterogeneous clusters. Roughly, Theorem 4.4 says that one can find an optimal policy in two steps. First, the customer must decide which epochs of which jobs get sent to each cluster. This is determined by picking $p_{ij}^{(h)}$, which we define to be the fraction of type-i jobs in epoch j that run on cluster h. Second, each cluster examines its assigned workload and solves for the optimal fixed-widths. Let $\{k_{ij}^{(h)}\}$ denote the number of type-h GPUs assigned to each class-i, epoch-j job. We solve both of these steps via a single convex optimization problem.

Theorem 4.4 (BOA in heterogeneous systems)**.** *In the heterogeneous setting without rescaling overheads, the budget-optimal allocation (BOA) policy is optimal and can be computed by solving the optimization problem:*

$$\begin{aligned} \underset{\{k_{ij}^{(h)}, p_{ij}^{(h)}\}}{\text{minimize}} \quad & \text{average } \mathcal{JCT} \\ \text{subject to} \quad & \text{operating budget} \le b, \\ & k_{ij}^{(h)} \ge \mathcal{K}_i^{(h)}, \\ & \sum_h p_{ij}^{(h)} = 1 \quad \forall i, j \end{aligned}$$

where

$$\text{average } \mathcal{JCT} = \frac{1}{\lambda} \sum_{i,j,h} \lambda_i \, p_{ij}^{(h)} \left(\frac{\mathbb{E}[X_{ij}]}{s_{ij}^{(h)}(k_{ij}^{(h)})} \right)$$

$$\text{operating budget} = \sum_{i,j,h} c^{(h)} \, \lambda_i \, p_{ij}^{(h)} \, k_{ij}^{(h)} \left(\frac{\mathbb{E}[X_{ij}]}{s_{ij}^{(h)}(k_{ij}^{(h)})} \right).$$

This is a convex optimization problem.

Theorem 4.4 follows directly from the proofs in Appendix A. To implement this policy, each epoch can be randomly routed to cluster h with probability $p_{ij}^{(h)}$.

4.3 Adding Rescaling Costs to BOA

In the idealized models, allocations can be changed instantaneously and without overhead. However, we must evaluate BOA in the realistic setting where changing a job's allocation incurs a *rescaling overhead* (see Sections 5.2 and 6). While a job is being rescaled, it consumes GPU resources but accumulates no training progress, increasing both the average JCT and the cost. Hence, we must adapt our results on BOA to handle rescaling overheads.

We begin in Theorem 4.5 by computing BOA, the optimal fixed-width policy, in a homogeneous cluster with rescaling overheads.

Theorem 4.5 (JCT and budget under rescaling)**.** *In the homogeneous setting with rescaling overheads, the optimal fixed-width policy, BOA, can be computed by solving the optimization problem*

$$\begin{aligned} \underset{\{k_{ij}\}}{\text{minimize}} \quad & \text{average } \mathcal{JCT} \\ \text{subject to} \quad & \text{operating budget} \le b, \\ & k_{ij} \ge \mathcal{K}_i \end{aligned} \tag{3}$$

where

$$\text{average } \mathcal{JCT} = \frac{1}{\lambda} \sum_i \lambda_i \sum_j \left(\frac{\mathbb{E}[X_{ij}]}{s_{ij}(k_{ij})} + r_i \cdot \mathbf{1}_{ij} \right),$$

$$\text{operating budget} = \sum_i \lambda_i \sum_j k_{ij} \left(\frac{\mathbb{E}[X_{ij}]}{s_{ij}(k_{ij})} + r_i \cdot \mathbf{1}_{ij} \right),$$

and $\mathbf{1}_{ij} = 1$ *if* $k_{ij} \neq k_{i(j-1)}$ *or* $j = 0$*, and* 0 *otherwise.*

Proof. The proof of this theorem is straightforward. We begin with the optimization problem from Theorem 4.3, but introduce a *rescaling indicator*, $\mathbf{1}_{ij}$, in both the objective and the constraint. This indicator is used to add a rescaling time when a job's allocation changes between adjacent epochs. □

While we can no longer prove that a fixed-width policy is optimal given rescaling overheads, the above BOA policy still represents the best possible fixed-width policy.

Note that, unlike the convex optimization problem from Theorem 4.3, (3) is a mixed-integer convex program (MICP). In practice, this MICP can be solved quickly in our implementation (see Section 5.2). However, for scalability to extremely large clusters, we also develop a heuristic for solving (3) that only invokes a sequence of convex optimization problems (see Appendix B). The policy derived by our heuristic solution performs as well as the true optimal solution in practice.

Rescaling overheads in heterogeneous clusters. In heterogeneous systems without rescaling overheads, both the average JCT and the operating budget depend only on the choices of $k_{ij}^{(h)}$ and $p_{ij}^{(h)}$, so any policy that realizes a given $\{p_{ij}^{(h)}\}$ is equally optimal. This enables the use of a simple, randomized scheme for routing work to each cluster.

However, given rescaling overheads, there are many ways to achieve a particular $p_{ij}^{(h)}$, each with potentially different overheads. Rather than search over the full space of policies that realize a given $\{p_{ij}^{(h)}\}$, we adopt a simplification: we randomly route each job to a particular cluster and run the job on the same cluster for its lifetime. That is, we choose values $p_i^{(h)}$ that determine the fraction of type-i jobs sent to cluster h. In our evaluations, this simplification incurs almost no loss of performance compared to searching the full space of routing probabilities (Appendix D). Using this simplification, we now define the BOA policy for a heterogeneous system with rescaling overheads.

Definition 4.6 (Heterogeneous BOA with rescaling)**.** The BOA policy for a heterogeneous system with rescaling overheads is defined as follows: At the arrival of any type i job, it is routed to the type-h GPU cluster with probability $p_i^{(h)}$ and it stays in that cluster throughout its lifetime; within each cluster, a fixed-width policy with parameters $k_{ij}^{(h)}$ is adopted. The parameters are the solution to the following optimization problem:

$$\begin{aligned} \underset{\{p_i^{(h)}, k_{ij}^{(h)}\}}{\text{minimize}} \quad & \textit{average JCT} \\ \text{subject to} \quad & \textit{operating budget} \le b, \\ & \sum_{h \in H} p_i^{(h)} = 1 \quad \forall i, \\ & k_{ij}^{(h)} \ge \mathcal{K}_i^{(h)}, \quad p_i^{(h)} \in [0,1], \end{aligned} \tag{4}$$

where

$$\text{average JCT} = \frac{1}{\lambda} \sum_{i,h} \lambda_i \, p_i^{(h)} \sum_j \left(\frac{\mathbb{E}[X_{ij}]}{s_{ij}^{(h)}(k_{ij}^{(h)})} + r_i \cdot \mathbf{1}_{ij}^{(h)} \right),$$

$$\text{op. budget} = \sum_{i,h} c^{(h)} \lambda_i \, p_i^{(h)} \sum_j k_{ij}^{(h)} \left(\frac{\mathbb{E}[X_{ij}]}{s_{ij}^{(h)}(k_{ij}^{(h)})} + r_i \cdot \mathbf{1}_{ij}^{(h)} \right),$$

and $\mathbf{1}_{ij}^{(h)} = 1$ if $k_{ij}^{(h)} \neq k_{i(j-1)}^{(h)}$ or $j = 0$, and 0 otherwise.

Once again, (4) is an MICP that can be solved quickly in practice (see Section 5.2). This policy forms the core of the BOA Constrictor system that we describe and evaluate in Sections 5 and 6, respectively.

5 System Design

In this section, we describe BOA Constrictor, our scheduler that realizes the BOA policy.

5.1 Implementing the BOA Policy

We implemented BOA Constrictor using the AdaptDL [1] scheduling framework. As shown in Figure 3, AdaptDL provides an interface to schedule and rescale training jobs according to an arbitrary scheduling policy. The AdaptDL control pods include a component for learning job speedup functions via profiling as each job runs. BOA Constrictor (marked in red) consumes these learned speedup functions. We also modified the AdaptDL control pods so that a scheduling policy has fine-grained control over the cluster size at every moment in time. This allows BOA Constrictor to control both job allocations and the overall cluster size at every moment in time. For a more detailed description of the changes we made to AdaptDL, see Appendix G.

Under BOA, the widths $\{k_{ij}^{(h)}\}$ are determined by the optimization problem (4). Once these widths are computed, the scheduler only needs to look up and execute the precomputed widths for each job epoch. BOA Constrictor exploits this property by decoupling the computation of $\{k_{ij}^{(h)}\}$ from the main scheduling loop: The *BOA Constrictor Width Calculator* computes and updates the widths $\{k_{ij}^{(h)}\}$ asynchronously while the synchronous scheduling logic executes the current $\{k_{ij}^{(h)}\}$ (see Figure 3 for an illustration).

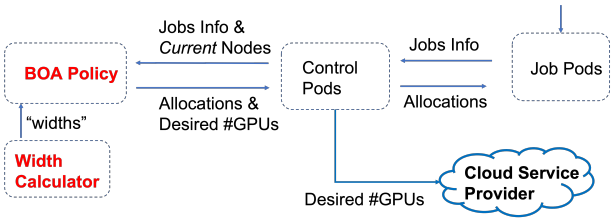


Figure 3. The BOA Constrictor architecture. The BOA Constrictor Width Calculator asynchronously refreshes $\{k_{ij}^{(h)}\}$, while the online scheduler executes the fixed-width policy.

Asynchronous width computation. The widths $\{k_{ij}^{(h)}\}$ themselves are produced by the *BOA Constrictor Width Calculator*, which solves the MICP (4). The Width Calculator runs as a background process and refreshes $\{k_{ij}^{(h)}\}$ periodically (every 15 minutes in our deployment). This decoupling keeps computationally intensive optimization off the scheduler's critical path. Furthermore, as the underlying profiler accumulates more runtime data, the Width Calculator incorporates refined estimates of the job speedup functions when re-solving the optimization without blocking job execution.

Synchronous fixed-width execution. The BOA policy maintains a table of the current widths. At each scheduling interval, it consults this table to make two decisions:

1. **Job allocation.** For each cluster, h, class-i jobs in epoch j are assigned $k_{ij}^{(h)}$ GPUs. The system must then solve the *placement problem* of selecting $k_{ij}^{(h)}$ GPUs for each job. We solve the placement problem via a greedy heuristic: Jobs already running on the correct number of GPUs keep their allocations. Additionally, where possible, the GPUs allocated to each job are condensed onto the smallest possible number of nodes. This is similar to the placement approach in Sia [16].
2. **Cluster sizing.** The desired cluster size is the sum of $k_{ij}^{(h)}$ across all active jobs. This sum is converted to a number of nodes demanded and forwarded to the AdaptDL Cluster Expander, which provisions or releases cloud instances.

5.2 Scheduler Overheads

BOA Constrictor incurs low scheduling overhead. The synchronous scheduler performs a simple table lookup of the $\{k_{ij}^{(h)}\}$ for the active jobs, which takes $<$ 1 ms per scheduling cycle. The BOA Constrictor Width Calculator runs asynchronously every 15 minutes. Solving the MICP (4) fits comfortably within this time window: In our evaluation with 5 job types (see Section 6.1), the MICP for a homogeneous

cluster terminates in 4 seconds and the MICP for a heterogeneous cluster with 3 GPU types terminates in 52 seconds.[4]

Although the MICP overhead is small in our experiments, our implementation in Section 6.3 uses a heuristic (see Appendix B) that approximates the MICP via a sequence of convex optimizations. This heuristic performs well in practice and scales to workloads much larger than those we evaluate.

Rescaling overheads. Although our scheduling policy itself has low overhead, the rescaling overheads in our system are non-negligible. Job rescaling overheads include checkpointing, pod termination, and restarting on new nodes. Because we leverage the underlying AdaptDL [1] rescaling mechanism, the overhead of a single rescaling is unchanged in our implementation. Specifically, for the example of a CIFAR-10 training job, we measured rescaling latencies of $\approx$ 20 sec on a "warm" machine (where container images are cached) and $\approx$ 120 sec on a "cold" machine. Of the 120-sec cold-start latency, $\approx$ 75 sec is spent installing Python packages and initializing CUDA contexts, followed by 25 sec for loading the dataset from the Elastic File System (EFS) to local storage. In the warm-start scenario, where these initialization steps can be skipped, half of the latency is due to establishing connections between distributed workers.

As we will show in Section 6, rescaling occasionally over the course of a job's lifetime can greatly improve the cost-performance tradeoff, while rescaling too frequently can easily inflate job completion times. BOA Constrictor is designed to rescale only when it is beneficial to do so.

6 Evaluation

We evaluate BOA Constrictor in three settings (Section 6.1). We perform simulations (Section 6.2) validated by deploying BOA Constrictor on AWS (Section 6.3). We also perform a sensitivity analysis of BOA Constrictor (Section 6.4).

6.1 Experimental Setup

We evaluate BOA Constrictor in three cluster settings targeted by three state-of-the-art adaptive schedulers. Many non-adaptive approaches [13, 24, 25] are dominated by the adaptive systems we consider, so we omit them from our evaluation. The settings we consider are as follows:

The Pollux [27] setting considers a homogeneous cluster where job batch size and learning rate are tuned adaptively.

The Rubick [34] setting considers a homogeneous cluster where each job's *parallel-training configuration* (e.g., data parallelism vs model parallelism) is chosen adaptively. In the Rubick paper, a job's training progress is measured using raw throughput, without considering statistical efficiency (see Section 2.1). We follow the same convention in this setting.

The Sia [16] setting considers a heterogeneous cluster where batch-size is tuned adaptively.

[4]Measured on a single CPU core at 5% optimality tolerance.

We compare against the schedulers proposed in [16, 27, 34] corresponding to each setting. These schedulers all rely on the same underlying heuristic for scheduling: maximizing the *cluster efficiency*. We define the cluster efficiency at any time as the sum of all running jobs' *normalized speedups* divided by the cluster size, where the normalized speedup of a class-i job in epoch j allocated k GPUs is given by

$$\tilde{s}_{ij}(k) \;=\; \frac{s_{ij}(k)}{s_{ij}(\mathcal{K}_i)}.$$

In the heterogeneous setting, we apply the same definition per GPU type, with $\mathcal{K}_i^{(h)}$ and $s_{ij}^{(h)}$ replacing $\mathcal{K}_i$ and s_{ij}.

Implementing the autoscaling variants. In each of the above settings, the proposed scheduling policy targets a fixed-size cluster. Hence, we additionally implement an autoscaling variant of each policy to compare to BOA Constrictor. An autoscaling variant of Pollux is proposed in [27], but is not implemented or evaluated. The proposed Pollux variant scales the cluster to meet a target level of cluster efficiency. The concept is that it is inefficient to allocate a high number of GPUs per job, so the cluster efficiency drops when there are few jobs in the system. Hence, the system can set a target level of cluster efficiency, and the cluster size can be scaled up or down depending on how many jobs are in the system to meet this target cluster efficiency. This proposal is evaluated in [27] only via a simulation of a *single job* whose speedup function changes over time. AdaptDL also provides a prototype of this proposal, but it does not scale to realistic cluster sizes or workloads. We implemented *Pollux-with-Autoscaling* using AdaptDL, a significant engineering effort.

Since all three competitor systems aim to maximize cluster efficiency, we extend the same autoscaling idea to Rubick and Sia, creating *Rubick-with-Autoscaling* and *Sia-with-Autoscaling* (see Appendix F for details). While these autoscaling variants are somewhat impractical due to the overhead of computing the cluster size hitting the target cluster efficiency, we ignore these overheads in our simulation experiments.

Workloads. Table 1 summarizes the traces used in each setting. The Pollux and Sia settings both use the production trace **newTrace** from [16]. Here, job runtimes vary by 10× across types, and interarrival times have squared coefficient of variation $C^2 = 2.65$. The Rubick setting uses the **Rubick** workload from [34]. Here, jobs train one of the seven Transformer-based models, two of which (LLaMA-2-7B and LLaMA-30B) cannot fit on a single GPU (i.e., $\mathcal{K}_i > 1$).

For the implementation experiment in Section 6.3, we subsample the original **workload-1** trace from [27] to create a trace that is cost-efficient to use on AWS. This trace consists of ResNet18, BERT, and DeepSpeech2 jobs, totaling 85 jobs.

Cluster hardware and cost. For each setting we configure our experiments to match the GPU types used in the corresponding paper:

Trace	Model	Dataset	Share of jobs
newTrace	ResNet18	CIFAR-10	50.42%
newTrace	DeepSpeech2	CMU-ARCTIC	23.54%
newTrace	BERT	SQuAD	21.67%
newTrace	YOLOv3	PASCAL-VOC	4.75%
newTrace	ResNet50	ImageNet	0.62%
Rubick	GPT-2	Wikipedia	17.00%
Rubick	BERT	Wikipedia	15.76%
Rubick	T5	Wikipedia	15.27%
Rubick	RoBERTa	WikiText-2	14.29%
Rubick	LLaMA-2-7B	WuDaoCorpora	13.30%
Rubick	LLaMA-30B	WuDaoCorpora	12.56%
Rubick	ViT	ImageNet-1K	11.82%

Table 1. Composition of the **newTrace** workload [16] (960 jobs total) and the **Rubick** workload [34] (406 jobs total).

The Pollux cluster is composed of NVIDIA T4 GPUs. Our AWS implementation in Section 6.3 uses the same GPU type via `g4dn.12xlarge` instances (4× T4 per node).
The Rubick cluster is composed of NVIDIA A800 GPUs (8× A100 per node, 100 Gb/s RDMA).
The Sia cluster was originally composed of three GPU types: NVIDIA T4, RTX 2080Ti, and V100. RTX 2080Ti is not offered by any major hyperscaler, so we cannot obtain a list price for it; we use NVIDIA A100's instead. The Azure SKU, memory tier, and prices we use for each type are listed in Appendix E.

Because the two homogeneous settings use a single GPU type, we report cost in *GPU-hours*. In the heterogeneous setting, we report cost in *USD per hour*.

6.2 Simulation Experiments

We simulate BOA Constrictor and the corresponding competitors in each of the three settings. For the Pollux and Sia settings, we use the simulator from [16]. For the Rubick setting, we use the simulator from [34]. Both simulators model each job from profiling data, account for rescaling overheads, and capture inter-job interference.

BOA Constrictor requires knowing a speedup function s_{ij} for each class i and epoch j. In each setting, the competitor policies already leverage a predicted speedup function to make their scheduling decisions, so BOA Constrictor uses the exact same predictions in our comparisons. These speedup function predictions are not exact, and they continue to evolve as more runtime data is collected. We seed each speedup prediction by assuming that one job of each type has been run prior to the start of our experiments.

Figure 4 shows the Pareto frontier of cost vs. performance for all three settings, with average JCT in the top row and P95 JCT in the bottom row used to measure performance. BOA Constrictor improves the Pareto frontier in every panel: across all three settings, BOA Constrictor achieves up to ∼1.8× lower average JCT than the strongest competitor at the same operating budget, and equivalently requires up to ∼2× less operating budget to achieve the same average JCT. One source of this improvement is that it is able to control the number of rescalings. For example, in the sia-setting simulation, when the budget is ∼135, on average each job rescales ∼4 times under Sia-with-Autoscaling, but only rescales < 1 time under BOA Constrictor.

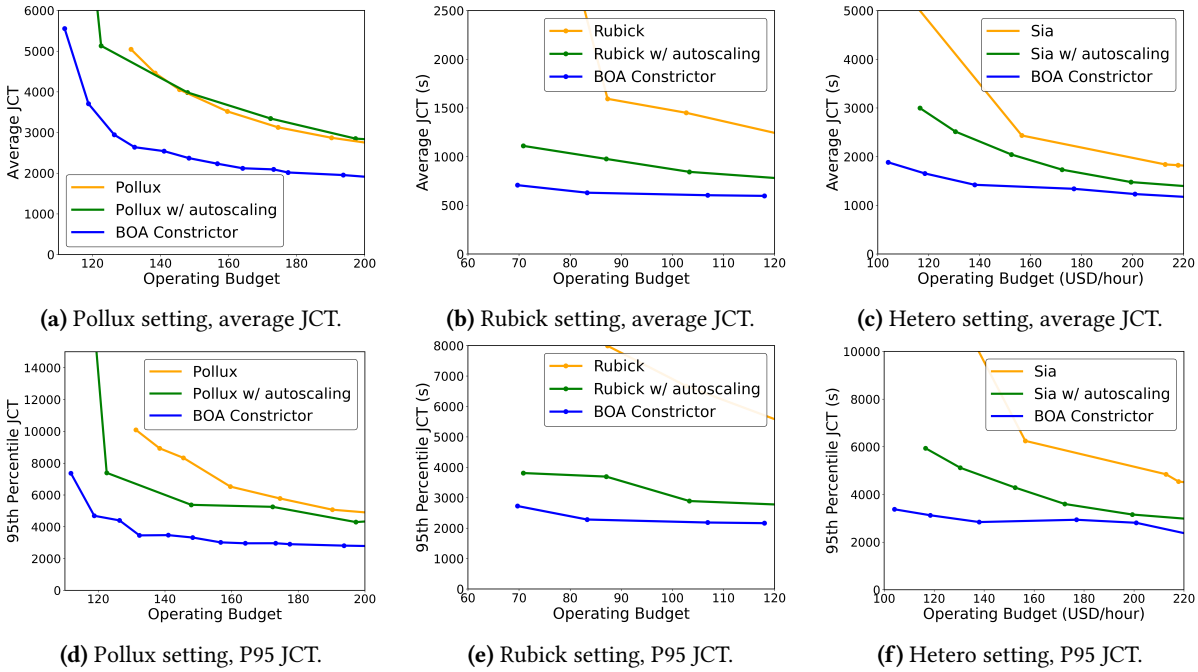
Although BOA Constrictor directly optimizes average JCT, it also benefits tail performance. For each of the competitor policies, a single unlucky job may be rescaled many times (e.g., due to a burst of arrivals). This cannot happen under BOA Constrictor, where each job's allocation is independent of the system state. As a result, BOA Constrictor reduces P95 JCT by up to ∼1.7× for a given operating budget.

6.3 Implementation Experiment

We deploy BOA Constrictor on an AWS cluster in the Pollux setting (see Section 6.1). Our real-world experiments both verify that BOA Constrictor runs as expected in a production-style cluster and validate our simulation results.
Measurement. The GPU usage statistics we report include the time from when a node first enters a "starting" state until it is marked for release by the allocation policy. In practice, there is some additional time from when a node is marked for release until it is fully reclaimed by the cloud provider. This additional time is controlled by the cloud provider's autoscaling infrastructure, varies from one cloud platform to the next, and cannot be directly optimized. We therefore exclude this reclamation time from our usage statistics. Appendix C shows that reclamation time affects all policies we consider, but has a smaller impact on BOA Constrictor than the competitor policies.
Workload and profiling. To limit the cost of running experiments on AWS, our implementation experiments use the shorter **workload-1** trace (see Section 6.1). The drawback here is that the scheduler has less time to learn job speedup functions and find good hyperparameter configurations for each job. To counteract this effect, we fix the predicted speedup functions and job hyperparameters in each experiment using profiling information gathered offline.

Hence, our implementation experiments indicate how each policy performs after running for long enough that the relevant performance models and hyperparameter selection algorithms have converged. These modifications help to isolate the performance gain that BOA Constrictor achieves by improving scheduling. While this simplification benefits Pollux significantly, BOA Constrictor is largely insensitive to errors in the predicted speedup function (see Figure 8).
Results. Figure 5 shows our implementation results alongside a matched simulation for a range of operating budgets.

The implementation results in Figure 5a show an improvement in average JCT at *all operating budgets*. BOA Constrictor reduces average JCT by up to a factor of ∼1.6× when both policies use a budget of ∼40 GPU-hours per hour, and reduces the budget required to achieve an average JCT of ∼2100 seconds by a factor of ∼2.2×. BOA Constrictor again



(a) Pollux setting, average JCT. **(b)** Rubick setting, average JCT. **(c)** Hetero setting, average JCT.

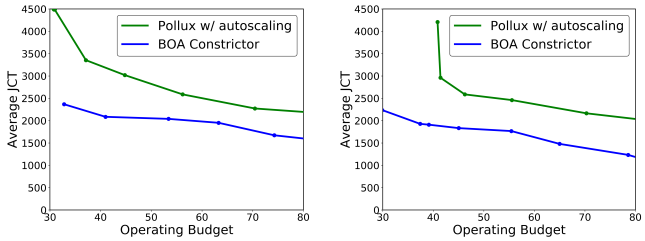
(d) Pollux setting, P95 JCT. **(e)** Rubick setting, P95 JCT. **(f)** Hetero setting, P95 JCT.

Figure 4. Comparing BOA Constrictor with the corresponding competitors in each of the three settings: Pollux setting (a, d), Rubick setting (b, e), and heterogeneous (Sia) setting (c, f). BOA Constrictor improves the Pareto frontier of both average JCT (top) and P95 JCT (bottom) in every setting.

benefits from finding the right GPU allocations with minimal rescaling: at a budget of ~55 GPU-hours per hour, BOA Constrictor rescales each job 3.54 times on average, compared to 6.71 times under the autoscaling variant of Pollux.

Crucially, the simulations in Figure 5b match the implementation results in Figure 5a. There are minor discrepancies between the two sets of experiments due to increased variability in the real-world experiments, which makes both policies' speedup models less accurate. Because this variability affects both policies similarly, the percentage difference between the policies in implementation is predicted closely by simulation. This validates our simulations from Section 6.2.

The experiment for Pollux-with-Autoscaling with the lowest budget encountered issues where Pollux's optimizer struggled to find allocations that met the policy's target cluster efficiency. The resulting policy rescaled jobs too frequently, leading to crashes when there were many jobs in the system. For this experiment, we report the average JCT measured at the time of the crash. Pollux's optimizer also failed to maintain the target cluster efficiency during low-budget simulation experiments. This suggests that autoscaling based on cluster efficiency is fundamentally limited in its ability to achieve very low resource utilization, whereas BOA Constrictor remains stable even under an extremely low budget.



(a) Implementation. **(b)** Simulation.

Figure 5. Comparing BOA Constrictor and Pollux-with-Autoscaling running 85 jobs from the **workload-1** trace [27]. Simulation results match implementation results, showing that BOA Constrictor significantly improves the Pareto frontier of average JCT vs. operating budget.

6.4 Sensitivity Analysis

We complete our evaluation by analyzing the robustness of the above results. For simplicity, we mainly focus on the Pollux setting for the sensitivity experiments below.

Is BOA Constrictor close to "cluster-efficiency-based autoscaling"? One potential concern is whether the decisions made by BOA Constrictor are close enough to the decisions made by Pollux that one could match BOA Constrictor's performance by simply tuning the parameters of Pollux to avoid some additional overheads.

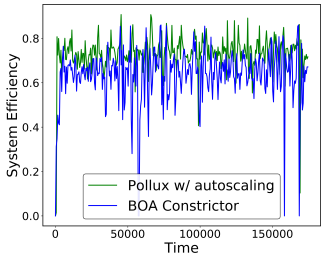


Figure 6. Cluster efficiency over time in the Pollux setting. BOA Constrictor maintains lower cluster efficiency than Pollux-with-Autoscaling, but provides better average JCT. This implies that cluster efficiency is a flawed heuristic.

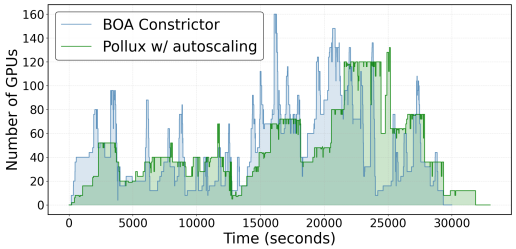


Figure 7. GPU usage of BOA Constrictor and Pollux-with-Autoscaling implementations. BOA Constrictor lowers JCTs by reacting faster and more aggressively to bursts in arrivals.

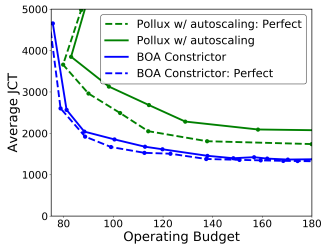


Figure 8. Average JCT for policies with perfect / imperfect speedup function information. BOA Constrictor is less sensitive to modeling accuracy than Pollux-with-Autoscaling.

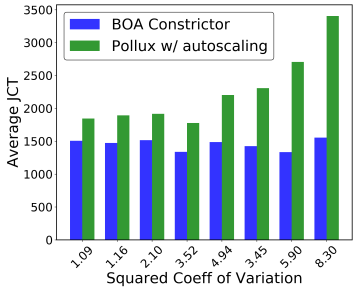


Figure 9. Comparison of the sensitivity of BOA Constrictor and Pollux-with-Autoscaling to arrival process variability. Here, both policies use a budget of ~120 GPUs to process a synthetic trace based on **newTrace**.

On the contrary, we find that BOA Constrictor and Pollux-with-Autoscaling pursue fundamentally different goals. Figure 6 plots cluster efficiency over time for the Pollux-setting simulation, comparing BOA Constrictor and Pollux-with-Autoscaling under the same operating budget. Pollux-with-Autoscaling runs at an average cluster efficiency of 0.73 while BOA Constrictor runs at an average cluster efficiency of 0.64. That is, BOA Constrictor uses the same number of GPU-hours on average, but achieves significantly lower average JCT by purposely using its resources *less efficiently* than Pollux-with-Autoscaling. This implies that the fundamental approach of Pollux-with-Autoscaling — pick a cluster size, maximize cluster efficiency given this cluster size — is flawed. The BOA policy makes allocation decisions that are not easily described in terms of cluster efficiency.

A more intuitive view of the two policies' autoscaling behavior comes from inspecting their GPU usage over time in the implementation experiment of Section 6.3. Figure 7 shows the GPU usage of the two policies given a budget of ~55 GPU-hours per hour. BOA Constrictor's advantage comes from the form of the BOA policy. The BOA policy computes the optimal allocation for each job epoch and decides ahead of time which rescaling costs are worth paying for each job. By contrast, Pollux-with-Autoscaling makes allocation decisions in a reactive manner and uses a fixed scheduling quantum of 60 seconds to avoid rescaling too frequently. As a result, BOA Constrictor reacts faster to bursts in arrivals. Pollux-with-Autoscaling allows queueing and scales the cluster size based on target efficiency rather than the optimal GPU demands of each job. Once Pollux-with-Autoscaling does react to a burst, it tends to not scale the cluster size as aggressively as BOA Constrictor. Both of these effects lead to higher JCTs for a given usage level.

How sensitive is BOA Constrictor to prediction error? Even when the models used to predict job speedups have converged, these models are still imperfect. We therefore evaluate BOA Constrictor's sensitivity to errors in the speedup models. Figure 8 shows via simulation that BOA Constrictor is significantly more robust to modeling errors than Pollux-with-Autoscaling. Here, we compare each policy's performance when using an imperfect speedup model to its performance when given perfect profiling information about each job. While the performance of BOA Constrictor is essentially unchanged by prediction error, the average JCT under Pollux-with-Autoscaling increases by up to a factor of ~1.4× in the face of prediction errors.

How sensitive are our results to workload variability? Another concern is whether our results hold as workload variability becomes more extreme. To examine this question,

we generate a synthetic trace based on **newTrace**. Our synthetic trace uses the same mixture of jobs, but we adjust the job arrival times to control how bursty the workload is. Specifically, we construct a time-varying Poisson process where jobs arrive at either a high rate or a low rate. To control variability, we fix the long-run average arrival rate, but create increasingly short and intense bursts of arrivals.

We describe the arrival variability via the squared coefficient of variation of interarrival times, C^2. Figure 9 shows that, as variability increases, BOA Constrictor outperforms Pollux-with-Autoscaling by an increasing margin. The increasing bursts in traffic amplify the differences between BOA Constrictor and Pollux-with-Autoscaling's slow, conservative autoscaling. For reference, **newTrace** has $C^2 = 2.65$.

7 Conclusion

Running ML training jobs in the cloud gives rise to a cost-performance tradeoff that the cloud customer must balance. Running on additional GPUs in parallel can reduce job completion times. However, because jobs receive a sublinear speedup from parallelism, this performance improvement comes at a price. We derive the BOA policy, which explicitly optimizes the average JCT across a stream of training jobs for any given budget constraint on GPU usage. BOA Constrictor significantly outperforms state-of-the-art schedulers by determining the correct allocations for each job proactively and avoiding excessive rescaling overheads. Our results apply broadly: BOA Constrictor optimizes the cost-performance tradeoff in homogeneous clusters, heterogeneous clusters, and for a wide range of distributed training workloads.

References

[1] 2026. Adaptdl Cluster Simulator: . https://github.com/petuum/adaptdl/tree/osdi21-artifact/simulator.

[2] 2026. Amazon, Google could see combined $50B in revenue from renting out GPUs. https://seekingalpha.com/news/4022837-amazon-google-50b-in-revenue-from-renting-out-gpus

[3] Benjamin Berg, Jan-Pieter Dorsman, and Mor Harchol-Balter. 2017. Towards optimality in parallel scheduling. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1, 2 (2017), 1–30.

[4] Benjamin Berg, Mor Harchol-Balter, Benjamin Moseley, Weina Wang, and Justin Whitehouse. 2020. Optimal resource allocation for elastic and inelastic jobs. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. 75–87.

[5] Benjamin Berg, Benjamin Moseley, Weina Wang, and Mor Harchol-Balter. 2024. Asymptotically optimal scheduling of multiple parallelizable job classes. *arXiv preprint arXiv:2404.00346* (2024).

[6] Benjamin Berg, Rein Vesilo, and Mor Harchol-Balter. 2020. heSRPT: Parallel scheduling to minimize mean slowdown. *Performance Evaluation* 144 (2020), 102–147.

[7] Benjamin Berg, Justin Whitehouse, Benjamin Moseley, Weina Wang, and Mor Harchol-Balter. 2022. The case for phase-aware scheduling of parallelizable jobs. *ACM SIGMETRICS Performance Evaluation Review* 49, 3 (2022), 65–66.

[8] Lisa Dunlap, Kirthevasan Kandasamy, Ujval Misra, Richard Liaw, Michael Jordan, Ion Stoica, and Joseph E Gonzalez. 2021. Elastic hyperparameter tuning on the cloud. In *Proceedings of the ACM Symposium on Cloud Computing*. 33–46.

[9] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. 2011. Distributed, robust auto-scaling policies for power management in compute intensive server farms. In *2011 Sixth Open Cirrus Summit*. IEEE, 1–5.

[10] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. 2012. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS)* 30, 4 (2012), 1–26.

[11] Samira Ghanbarian, Arpan Mukhopadhyay, Ravi R. Mazumdar, and Fabrice M. Guillemin. 2024. On Optimal Server Allocation for Moldable Jobs with Concave Speed-Up. In *Proceedings of the Twenty-Fifth International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing* (Athens, Greece) *(MobiHoc '24)*. 191–200.

[12] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. Elasticflow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 266–280.

[13] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 485–500.

[14] M. Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press.

[15] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2023. Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 457–472.

[16] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 642–657.

[17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[18] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. Allox: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[19] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research* 18, 185 (2018), 1–52.

[20] Mingzhen Li, Wencong Xiao, Hailong Yang, Biao Sun, Hanyu Zhao, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, et al. 2023. EasyScale: Elastic training with consistent accuracy and improved utilization on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[21] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 289–304.

[22] Ujval Misra, Richard Liaw, Lisa Dunlap, Romil Bhardwaj, Kirthevasan Kandasamy, Joseph E Gonzalez, Ion Stoica, and Alexey Tumanov. 2021. Rubberband: cloud-based hyperparameter tuning. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 327–342.

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

[24] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 481–498.

[25] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.

[26] Konstantinos Psychas and Javad Ghaderi. 2022. A theory of autoscaling for resource reservation in cloud services. *Stochastic Systems* 12, 3 (2022), 227–252.

[27] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*.

[28] Mathew Salvaris, Danielle Dean, and Wee Hyong Tok. 2018. Microsoft AI platform. *Deep Learning with Azure: Building and Deploying Artificial Intelligence Solutions on the Microsoft AI Platform* (2018), 79–98.

[29] Ganjihal Sanjeev and Alvarez-Parmar Re. 2023. Maximizing GPU utilization with NVIDIA's Multi-Instance GPU (MIG) on Amazon EKS: Running more pods per GPU for enhanced performance. https://aws.amazon.com/blogs/containers/maximizing-gpu-utilization-with-nvidias-multi-instance-gpu-mig-on-amazon-eks-running-more-pods-per-gpu-for-enhanced-performance/.

[30] Ziv Scully, Isaac Grosof, and Mor Harchol-Balter. 2020. The Gittins policy is nearly optimal in the M/G/k under extremely general conditions. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.

[31] David Tse and Pramod Viswanath. 2005. *Fundamentals of wireless communication*. Cambridge University Press.

[32] Sahil Tyagi and Prateek Sharma. 2023. Scavenger: A cloud service for optimizing cost and performance of ML training. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 403–413.

[33] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 533–548.

[34] Xinyi Zhang, Hanyu Zhao, Wencong Xiao, Xianyan Jia, Fei Xu, Yong Li, Wei Lin, and Fangming Liu. 2025. Rubick: Exploiting job reconfigurability for deep learning cluster scheduling. *Proceedings of Machine Learning and Systems* 7 (2025).

[35] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. 2023. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 703–723.

A Proof in offline setting

Throughout this appendix, we consider the setting of homogenous GPUs without rescaling overheads. Our main goal is to prove the following theorem, which derives the BOA policy in this setting.

Theorem A.1. *We define BOA-no-rescaling as the fixed-width policy whose widths $\{k_{ij}\}$ are given by the solution of*

$$\begin{aligned} \underset{\{k_{ij}\}}{\text{minimize}} \quad & \frac{1}{\lambda}\sum_{i,j}\frac{\rho_{ij}}{s_{ij}(k_{ij})} \\ \text{subject to} \quad & \sum_{i,j}\frac{\rho_{ij}\,k_{ij}}{s_{ij}(k_{ij})} \le b, \\ & k_{ij} \ge \mathcal{K}_i. \end{aligned} \tag{5}$$

BOA-no-rescaling is the budget-optimal allocation policy for homogenous clusters with no rescaling overhead, and this policy can be computed by solving a convex optimization problem.

This appendix proves Theorem A.1 via a series of lemmas. Our argument uses an offline auxiliary problem as a technical tool: in the offline problem, the arrival times and inherent work of every job at every epoch are known to the customer a priori. We characterize the optimal offline policy on every *well-behaved* sample path (defined below) and then lift this offline result to the online setting.

A.1 Offline auxiliary problem and well-behaved sample paths

A sample path $\mathcal{A}$ is an infinite sequence of arrival times and inherent works, where $x_{ij}^{(\ell)}$ denotes the inherent work of the ℓth arriving type-i job in epoch j. Let $n_i(t)$ denote the number of type-i arrivals by time t, and $n(t) := \sum_{i=1}^{M} n_i(t)$ the total number of arrivals.

Definition A.2 (Well-behaved sample path)**.** A sample path $\mathcal{A}$ is *well-behaved* if its time-average arrival rates and time-average inherent work converge to their finite means:

$$\lambda_i = \lim_{t\to\infty}\frac{n_i(t)}{t}, \qquad \mathbb{E}[X_{ij}] = \lim_{t\to\infty}\frac{\sum_{\ell=1}^{n_i(t)} x_{ij}^{(\ell)}}{n_i(t)} \quad \forall i, j.$$

The finite-means assumption stated in Section 3 implies that a random sample path is well-behaved with probability 1, so it suffices to derive a policy that is optimal for the set of well-behaved sample paths.

A.2 Structural lemmas

Lemma A.3 (No queueing)**.** *For any well-behaved sample path $\mathcal{A}$, no job queues under the optimal policy.*

Proof. Assume there is a job queueing in the optimal policy. Suppose the job is allocated $k(t)$ GPUs at time t. Then, we create a policy π' that removes all the queueing from the optimal policy. Mathematically, at any time t, π' allocates the job $k(t')$ number of GPUs where t' is the smallest time that satisfies $t = t' - q(t')$, where $q(t')$ is the jobs' total queueing time under the optimal policy before time t'.

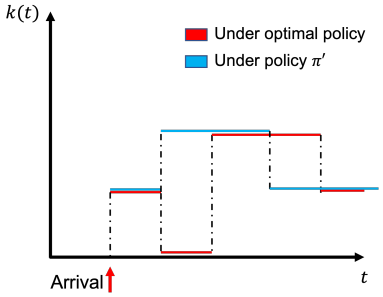


Figure 10. Illustration for the construction of policy π'.

Removing its waiting time makes the mean response time lower but leaves the total GPU-hours the same, a contradiction. Note that for the online problem, we cannot create such a policy π' because we do not know what happens in the future. □

Lemma A.4 (Same allocation for same speedup)**.** *Under any well-behaved sample path $\mathcal{A}$, any type-i job at epoch j is always assigned the same number of GPUs, and this allocation does not change until the epoch completes.*

Proof. Suppose the optimal policy P does not satisfy the statement in the lemma. Then either P assigns $k_1 \neq k_2$ GPUs to different type-i jobs at epoch j, or P assigns $k_1 \neq k_2$ GPUs to a single type-i job at epoch j at different times. Let x_1, x_2 be the work completed using k_1 and k_2 GPUs, respectively.

We construct a policy P' which uses the same GPU-hours, but has lower mean response time than P. P' is identical to P except for the work x_1 and x_2: Instead of using two different assignments k_1 and k_2, P' will choose a constant number of GPUs, k, to use in both instance. Let $t_1 = \frac{x_1}{s_{ij}(k_1)}$ and $t_2 = \frac{x_2}{s_{ij}(k_2)}$ be the durations of each of the GPU assignments under P. We choose k to be the time average of the two assignments by setting

$$k = k_1 \cdot \frac{t_1}{t_1+t_2} + k_2 \cdot \frac{t_2}{t_1+t_2}. \tag{6}$$

The concavity of $s_{ij}(k)$ implies that the total time to complete x_1 and x_2 is lower under P'. To see this, consider the average rate of work completion when processing x_1 and x_2 under both policies. The average work rate under P is

$$\frac{t_1}{t_1+t_2}\cdot s_{ij}(k_1) + \frac{t_2}{t_1+t_2}\cdot s_{ij}(k_2).$$

The average work rate under P' is $s_{ij}(k)$. By concavity, we have

$$\frac{t_1}{t_1+t_2}\cdot s_{ij}(k_1)+\frac{t_2}{t_1+t_2}\cdot s_{ij}(k_2) \le s_{ij}\left(\frac{t_1}{t_1+t_2}\cdot k_1+\frac{t_2}{t_1+t_2}\cdot k_2\right)=s_{ij}(k).$$

The total time to process x_1 and x_2 can be computed as the total work (x_1+x_2) divided by the average work rate. We thus have that P' completes the x_1+x_2 work sooner than P. As a result, P' has a lower mean response time than P.

It is easy to see that P' does not use more total GPU-hours than P. Specifically, note that P uses $k_1t_1+k_2t_2$ GPU-hours to process x_1 and x_2. Let t_1' and t_2' be the times required to process x_1 and x_2 respectively under P'. Then the GPU-hours used to process x_1 and x_2 under P' is

$$(t_1'+t_2')\left(\frac{k_1t_1}{t_1+t_2}+\frac{k_2t_2}{t_1+t_2}\right).$$

We have already shown that $t_1'+t_2'\le t_1+t_2$, giving

$$(t_1'+t_2')\left(\frac{k_1t_1}{t_1+t_2}+\frac{k_2t_2}{t_1+t_2}\right) \le (t_1+t_2)\left(\frac{k_1t_1}{t_1+t_2}+\frac{k_2t_2}{t_1+t_2}\right)=k_1t_1+k_2t_2.$$

This leads to the contradiction, in that the policy P' is using less budget but achieving better mean response time. □

Lemmas A.3 and A.4 together imply that the optimal offline policy is a *fixed-width* policy in the sense of Definition 4.1: every job is allocated some number of GPUs immediately upon arrival, and its allocation depends only on its class and current epoch.

A.3 Operating budget of fixed-width policies

The following lemma develops an alternate formulation of the operating budget of a fixed-width policy and matches the closed-form expressions used in Section 4.1.

Lemma A.5 (Operating budget of a fixed-width policy)**.** *Given a well-behaved sample path $\mathcal{A}$, the operating budget of a fixed width policy with parameters $\{k_{ij}\}$ is*

$$\bar{B}:=\lim_{t\to\infty}\frac{\int_0^t K(s)ds}{t}\overset{(a)}{=}\lim_{t\to\infty}\frac{\sum_{i=1}^{n(t)}B^{(i)}}{t}\overset{(b)}{=}\sum_{i,j}\frac{\rho_{ij}k_{ij}}{s_{ij}(k_{ij})}.$$

Here $B^{(i)}$ is defined to be the GPU-hours used to complete the i^{th} arriving job.

Proof. Part (a) of our claim says that tracking the total GPU usage at every time t is equivalent to tracking the GPU-hours used to process each job, $B^{(i)}$.

To prove part (b), we show that we can take the limit of this equivalent formulation to prove our claim. Note that the fixed width policy assigns k_i GPUs to any type-i job. Hence, the GPU-hours spent on the ℓ^{th} type-i job is $\sum_j \frac{x_{ij}^{(\ell)}k_{ij}}{s_{ij}(k_{ij})}$.

Thus we have that

$$\lim_{t\to\infty}\frac{\sum_{i=1}^{n(t)}B^{(i)}}{t}=\lim_{t\to\infty}\frac{\sum_{i=1}^{M}\sum_{\ell=1}^{n_i(t)}\sum_j\frac{x_{ij}^{(\ell)}k_{ij}}{s_{ij}(k_{ij})}}{t}=\sum_{i=1}^{M}\sum_j\frac{k_{ij}}{s_{ij}(k_{ij})}\left(\lim_{t\to\infty}\frac{\sum_{\ell=1}^{n_i(t)}x_{ij}^{(\ell)}}{t}\right),$$

where

$$\lim_{t\to\infty}\frac{\sum_{\ell=1}^{n_i(t)}x_{ij}^{(\ell)}}{t}=\lim_{t\to\infty}\frac{\sum_{\ell=1}^{n_i(t)}x_{ij}^{(\ell)}}{n_i(t)}\frac{n_i(t)}{t}=\lambda_i\mathbb{E}[X_{ij}]=\rho_{ij}.$$

□

A.4 Optimality of BOA-no-rescaling and lift to the online setting

We now show that BOA-no-rescaling is offline optimal for any well-behaved sample path $\mathcal{A}$.

Lemma A.6. *For any well-behaved sample path $\mathcal{A}$, BOA-no-rescaling is the optimal offline policy.*

Proof. Lemmas A.3 and A.4 show that the optimal offline policy is a fixed-width policy. Thus, it suffices to show that BOA-no-rescaling is the optimal offline fixed-width policy.

For any job in $\mathcal{A}$ of type i and size x_{ij} at epoch j, the JCT under a fixed width policy parameterized with k_i is $\sum_j \frac{x_{ij}}{s_{ij}(k_{ij})}$. Thus, we have that

$$\mathbb{E}[T]:=\lim_{t\to\infty}\frac{\sum_{i=1}^{n(t)}T_i}{n(t)}=\lim_{t\to\infty}\sum_{i=1}^{M}\frac{\sum_{\ell=1}^{n_i(t)}\sum_j x_{ij}^{(\ell)}}{s_{ij}(k_{ij})n(t)}=\sum_{i=1}^{M}\sum_j\frac{1}{s_{ij}(k_{ij})}\left(\lim_{t\to\infty}\frac{\sum_{\ell=1}^{n_i(t)}x_{ij}^{(\ell)}}{n(t)}\right)=\frac{1}{\lambda}\sum_{i=1}^{M}\frac{\sum_j\rho_{ij}}{s_{ij}(k_{ij})}.$$

Moreover, by Lemma A.5, the operating budget is $\sum_{i,j}\frac{\rho_{ij}k_{ij}}{s_{ij}(k_{ij})}$. Thus solving the optimal set of k_{ij} is equivalent to solving the following optimization problem:

$$\begin{aligned}\underset{k_{ij}}{\text{minimize}}\quad & \frac{1}{\lambda}\sum_{i,j}\frac{\rho_{ij}}{s_{ij}(k_{ij})}\\ \text{subject to}\quad & \sum_{i,j}\frac{\rho_{ij}k_{ij}}{s_{ij}(k_{ij})}\le b.\end{aligned}$$

This is the optimization problem (5) that defines BOA-no-rescaling. This shows that BOA-no-rescaling is the optimal fixed-width policy. □

Lift to the online setting. We now complete the proof of Theorem 4.3 by lifting Lemma A.6 to the online problem. The optimization (5) that defines BOA-no-rescaling depends only on the load parameters $\{\rho_{ij}\}$, which are known to the customer in the online setting. Hence BOA-no-rescaling is

implementable online. By Lemma A.6, BOA-no-rescaling is optimal on every well-behaved sample path; since the finite-means assumption implies that the set of sample paths that are not well-behaved has measure zero, BOA-no-rescaling is optimal in the online setting almost surely.

A.5 Convex optimization Translation

In this appendix, we rewrite the optimization problem (5) into a convex optimization problem. The problem (5) is not immediately convex because the constraint terms $\frac{k_{ij}}{s_{ij}(k_{ij})}$ are not necessarily convex. However, by applying a change of variables, we can translate it into a convex optimization problem.

Theorem A.7. *The optimization problem* (5) *can be solved in the following two steps:*

1. *Solve the convex optimization problem* (7) *to get the optimal solution* z_{ij}*;*
2. *Get the optimal* $k_{ij} = s_{ij}^{-1}(\frac{1}{z_{ij}})$ *where* s_{ij}^{-1} *is the inverse of the speedup function* s_{ij}*.*

Proof. Define $z_{ij} := \frac{1}{s_{ij}(k_{ij})}$. Define $d_{ij} := \sup_{k \geq \mathcal{K}_i} s_{ij}(k)$ (if it does not exist, let $d_{ij} = \infty$). If exists k such that $s_{ij}(k) = d_{ij}$, denote the smallest one by ξ_{ij}; Otherwise, let ξ_{ij} be $\perp$.

Note that since s_{ij} is non-decreasing and concave, we have that it is strictly increasing in $[\mathcal{K}_i, \xi_{ij}]$ ($[\mathcal{K}_i, \infty)$ if $\xi_{ij} = \perp$). Thus we can define the function $\beta_{ij} = s_{ij}^{-1}$ to be the inverse of the speedup function on $[s_{ij}(\mathcal{K}_i), d_{ij}]$ ($[s_{ij}(\mathcal{K}_i), d_{ij})$ if $\xi_{ij} = \perp$). Using the fact that s_{ij} is increasing, positive and concave, we have that β_{ij} is decreasing and convex.

By definition of z_{ij}, we have that $k_{ij} = \beta_{ij}(\frac{1}{z_{ij}})$. Substituting this into the optimization problem (5), we have

$$\begin{aligned} \underset{z_{ij}}{\text{minimize}} \quad & \sum_{i,j} \rho_{ij} z_{ij} \\ \text{subject to} \quad & \sum_{i,j} \rho_{ij} z_{ij} \beta_{ij}(\frac{1}{z_{ij}}) \leq b \\ & \frac{1}{s_{ij}(\mathcal{K}_i)} \geq z_{ij} \geq \frac{1}{d_{ij}}. \end{aligned} \tag{7}$$

Note that

$$\frac{d}{dz_{ij}}(z_{ij}\beta_{ij}(\frac{1}{z_{ij}})) = \beta_{ij}(\frac{1}{z_{ij}}) - \frac{1}{z_{ij}}\beta'_{ij}(\frac{1}{z_{ij}}),$$

and

$$\begin{aligned} \frac{d^2}{dz_{ij}^2}(z_{ij}\beta_{ij}(\frac{1}{z_{ij}})) &= \frac{-1}{z_{ij}^2}\beta'_{ij}(\frac{1}{z_{ij}}) + \frac{1}{z_{ij}^2}\beta'_{ij}(\frac{1}{z_{ij}}) + \frac{1}{z_{ij}^2}\beta''_{ij}(\frac{1}{z_{ij}}) \\ &= \frac{1}{z_{ij}^2}\beta''_{ij}(\frac{1}{z_{ij}}) > 0. \end{aligned}$$

Thus the optimization problem (7) is a convex optimization, which we can numerically solve for the optimal z_{ij}.

Finally, given the optimal z_{ij}, we can get the optimal k_{ij} by letting $k_{ij} = \beta_{ij}(\frac{1}{z_{ij}})$.

□

B Detailed heuristic for BOA width calculator

We provide the detailed pseudocode for our BOA width calculator as follows (Algorithm 1). We note that we only implement the glue-parameter-based heuristic for homogeneous clusters (i.e., a heuristic for the MICP (3)). For the BOA policy on heterogeneous clusters, we directly use a solver to solve the MICP (4).

C Usage Statistics

As discussed in Section 6.1, the operating budget reported in Figure 5a reflects the effective GPU hours—the duration during which GPUs are actively allocated to jobs. This metric excludes the platform-dependent latency between when a node is marked for release by the allocator and when it is actually reclaimed by the cloud platform.

Figure 11 extends Figure 5a by providing a comparison of the total usage for both policies. These results demonstrate that these infrastructure-level reclamation overheads affect the operating budget of both policies, but the negative effect on BOA Constrictor is smaller. Moreover, even with these infrastructure-level reclamation overheads, BOA Constrictor still significantly improves the Pareto frontier of average JCT vs. operating budget.

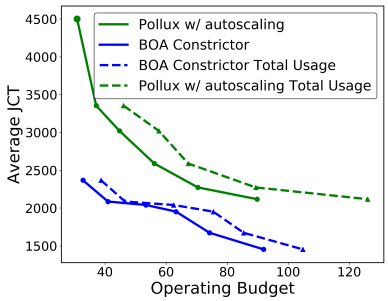


Figure 11. Actual Usage v.s. Effective Usage

D Loss of the arrival-routing simplification

Section 4.2 restricts attention to *hetero-fixed-width* policies (Definition 4.6). This appendix compares the optimal hetero-fixed-width policy to the optimal unrestricted per-epoch policy in the no-rescaling setting, showing that the simplification incurs negligible loss in average JCT.

The unrestricted optimization, parameterized by per-epoch routing $\{p_{ij}^{(h)}\}$ and widths $\{k_{ij}^{(h)}\}$, minimizes the average JCT

Algorithm 1: BOA Width Calculator

Input: Total budget b, Job classes $1, \ldots, M$ with epoch counts $l_1, \ldots, l_M$.
Output: Optimal parameters $\{k_{ij}^*\}$ for the fixed-width policy.

```
// First Step: Generate a set of glue configurations.
```

$\mathcal{G} \leftarrow \emptyset$
for $n \leftarrow 1$ **to** 50 **do**
  foreach *job class* $i \in \{1, \ldots, M\}$ **do**
    Define candidate set $S_i = \{2^0, 2^1, \ldots, 2^{\lfloor \log_2 l_i \rfloor}\}$
    Sample g_i uniformly from S_i
  end
  Add configuration $G = \{g_1, \ldots, g_M\}$ to $\mathcal{G}$
end

```
// Second Step: Solve a feasible solution for each glue configuration.
```

$\mathbb{E}[T]_{\min} \leftarrow \infty$
$\{k_{ij}^*\} \leftarrow \emptyset$
foreach $G \in \mathcal{G}$ **do**
  Construct super-epochs by gluing adjacent epochs for each class i according to $g_i \in G$
  $b_{run} \leftarrow b$
  $b' \leftarrow \infty$
  while $b' > b$ **do**
    Compute $\{k_{ij}\}$ by solving **Optimization Problem** (5) with budget b_{run}
    Round each k_{ij} to the nearest integer on the non-decreasing concave hull of s_{ij}
    Compute $\mathbb{E}[T]$ and total cost b' using $\{k_{ij}\}$ (Theorem 4.5)
    if $b' > b$ **then**
      $b_{run} \leftarrow 0.99 \cdot b_{run}$
    else
      if $\mathbb{E}[T] < \mathbb{E}[T]_{\min}$ **then**
        $\mathbb{E}[T]_{\min} \leftarrow \mathbb{E}[T]$
        $\{k_{ij}^*\} \leftarrow \{k_{ij}\}$
      end
    end
  end
end
return $\{k_{ij}^*\}$

subject to the operating cost being at most b:

$$\begin{aligned} \underset{\{k_{ij}^{(h)}, p_{ij}^{(h)}\}}{\text{minimize}} \quad & \frac{1}{\lambda} \sum_{i,j,h} \frac{p_{ij}^{(h)} \rho_{ij}}{s_{ij}^{(h)}(k_{ij}^{(h)})} \\ \text{subject to} \quad & \sum_{i,j,h} \frac{c^{(h)} p_{ij}^{(h)} \rho_{ij} k_{ij}^{(h)}}{s_{ij}^{(h)}(k_{ij}^{(h)})} \le b, \\ & \sum_{h \in H} p_{ij}^{(h)} = 1 \quad \forall i, j, \\ & k_{ij}^{(h)} \ge \mathcal{K}_i^{(h)},\ p_{ij}^{(h)} \in [0, 1]. \end{aligned} \tag{8}$$

The hetero-fixed-width optimization replaces $p_{ij}^{(h)}$ with per-class arrival routing $p_i^{(h)}$ (no j index):

$$\begin{aligned} \underset{\{k_{ij}^{(h)}, p_i^{(h)}\}}{\text{minimize}} \quad & \frac{1}{\lambda} \sum_{i,j,h} \frac{p_i^{(h)} \rho_{ij}}{s_{ij}^{(h)}(k_{ij}^{(h)})} \\ \text{subject to} \quad & \sum_{i,j,h} \frac{c^{(h)} p_i^{(h)} \rho_{ij} k_{ij}^{(h)}}{s_{ij}^{(h)}(k_{ij}^{(h)})} \le b, \\ & \sum_{h \in H} p_i^{(h)} = 1 \quad \forall i, \\ & k_{ij}^{(h)} \ge \mathcal{K}_i^{(h)},\ p_i^{(h)} \in [0, 1]. \end{aligned} \tag{9}$$

We compared the average JCT of the two above approaches at a variety of budget levels, and our simplification was always within 5% of the true optimal average JCT.

E Heterogeneous cluster pricing

Table 2 lists the Azure SKUs and on-demand list prices we use to compute USD-denominated operating budget in the heterogeneous Sia setting (Section 6.1).

GPU type	Memory	Azure SKU	USD / GPU-hour
T4	16 GB	NC64as_T4_v3	1.088
V100	32 GB	ND40rs_v2	2.754
A100	40 GB	ND96asr_v4	3.400

Table 2. On-demand prices used to compute USD-denominated cost in the heterogeneous Sia setting (Azure list prices, `eastus`, queried May 2026).

F Implementing the autoscaling variants

This appendix gives the full details of the three autoscaling baselines (Pollux-with-Autoscaling, Rubick-with-Autoscaling, and Sia-with-Autoscaling) summarized in Section 6.1. All three are built around the same cluster-efficiency heuristic that the corresponding fixed-cluster schedulers already use; the differences are in how that heuristic is operationalized when the cluster size becomes a decision variable.

F.1 Tuning the target efficiency c

While [27] only suggests setting the target efficiency at 0.5, we explore the full range of the target efficiency level, c; this is a performance knob that we can tune to evaluate the autoscaling variants at different operating budgets. Specifically, we define $\Delta = \min\{0.3(1-c), 0.3c\}$. The cluster size is increased whenever the cluster efficiency exceeds $c + \Delta$, and decreased whenever the cluster efficiency falls below $c - \Delta$. The new cluster size is then set using combinatorial optimization to select the cluster size and job allocations whose cluster efficiency is closest to c. Importantly, although adjusting the target efficiency c controls the GPU usage of

the policy, there is no analysis predicting the average number of GPU-hours that an autoscaling variant will consume for a given value of c.

F.2 Rubick: non-monotone cluster efficiency

In the Pollux setting, where every job fits on a single GPU, the cluster efficiency is monotone in the cluster size, and the cluster size matching a target efficiency c can be located by binary search. In the Rubick setting, however, this monotonicity fails. For example, suppose the workload contains one ordinary job and one $\mathcal{K}_i = 8$ job. At cluster size 4, the large job is queued and the cluster efficiency is determined entirely by the ordinary job; at cluster size 12, both jobs are running and the cluster efficiency becomes higher because the normalized speedup of the $\mathcal{K}_i = 8$ job is exactly 1. Because the cluster efficiency is no longer monotone in cluster size, the autoscaler cannot use binary search and must instead enumerate every feasible cluster size to find the one whose efficiency is closest to c. We tolerate this overhead in our simulator (the simulator pauses while the sweep completes) for the sake of a fair comparison; even so, BOA Constrictor outperforms Rubick-with-Autoscaling (Section 6.2).

F.3 Sia: which cluster to scale

In the heterogeneous Sia setting, when the cluster efficiency drifts away from its target, the heuristic does not specify which GPU-type cluster should be grown or shrunk; the adaptive-training literature on heterogeneous clusters provides no answer either. Our implementation maintains the same efficiency target for each GPU-type cluster and lets each cluster autoscale accordingly and independently. Even with this best-effort design, BOA Constrictor still outperforms Sia-with-Autoscaling (Section 6.2).

G Extending the AdaptDL Framework

We extend the AdaptDL framework to help evaluate BOA Constrictor and improve user experience.

Changes in Autoscaling Mechanism. AdaptDL relies on a Kubernetes autoscaler and uses *placeholder pods* to signal when the cluster should be sized up or down. We observe that this mechanism is too slow for the scaling decisions that BOA Constrictor wants to make, so we instead directly expose the AWS autoscaling API to BOA Constrictor.

Profile Sharing. The original AdaptDL framework trains a performance model from scratch for every new job. While not central to our implementation, we extended the system to allow jobs of the same class to share performance models. This is realized by a global profiler on top of the AdaptDL schedulers. The main purpose of this modification is to isolate our improvement in scheduling from the orthogonal hyperparameter tuning process (see Section 6.3).

Decision Support Tool. We leverage the predictability of the BOA policy to build an offline profiling calculator. Given a workload trace as input, BOA Constrictor can provide the full Pareto frontier by running Algorithm 1 at different budgets. This tool allows a customer to select an optimal budget b based on their desired average JCT before provisioning any real resources.

Extensibility to Hyperparameter Tuning. While we deploy the batch-size and learning-rate tuning used in [27] for each job in our evaluations, the BOA policy is compatible with *any* hyperparameter tuning approach. The BOA policy and the hyperparameter optimization solve orthogonal problems: the BOA policy optimizes system performance given the speedup functions, while the hyperparameter optimization improves those speedup functions. BOA Constrictor does not need to know the particular hyperparameter choices, so long as it can obtain the speedup functions.