George: Learning to Place Long-Lived Containers in Large Clusters with Operation Constraints

Suyi Li*
HKUST
slida@cse.ust.hk

Luping Wang*
HKUST, Alibaba Group
lwangbm@cse.ust.hk

Wei Wang
HKUST
weiwa@cse.ust.hk

Yinghao Yu
Alibaba Group, HKUST
yinghao.yyh@alibaba-inc.com

Bo Li
HKUST
bli@cse.ust.hk

Abstract

Online cloud services are widely deployed as Long-Running Applications (LRAs) hosted in containers. Placing LRA containers turns out to be particularly challenging due to the complex interference between co-located containers and the operation constraints in production clusters such as fault tolerance, disaster avoidance and incremental deployment. Existing schedulers typically provide APIs for operators to manually specify the container scheduling requirements and offer only qualitative scheduling guidelines for container placement. Such schedulers, do not perform well in terms of both performance and scale, while also requiring manual intervention.

In this work, we propose George, an end-to-end general-purpose LRA scheduler by leveraging the state-of-the-art Reinforcement Learning (RL) techniques to intelligently schedule LRA containers. We present an optimal container placement formulation for the first time with the objective of maximizing container placement performance subject to a set of operation constraints. One fundamental challenge in scheduling is to categorically satisfy different operation constraints in practice; specifically, to guarantee hard constraints and ensure soft constraints violations within a pre-defined threshold. We design a novel projection-based proximal policy optimization (PPPO) algorithm in combination with an Integer Linear optimization technique to intelligently schedule LRA containers under operation constraints. In order to reduce the training time, we apply transfer learning technique by taking advantage of the similarity in different LRA scheduling events. We prove theoretically that our proposed algorithm is effective, stable, and safe. We implement George as a plug-in service in Docker Swarm. Our in-house cluster demonstrates that George can maximize the LRA performance while enforcing the hard constraints and the soft constraints with a pre-defined threshold. The experiments show that George improves LRA performance and scale tremendously by requiring less than 1 hour scheduling time in a large cluster with 2K containers and 700 machines, 16× faster than existing schedulers. Compared with state-of-the-art alternatives, George also achieves 26% higher container performance with up to 70% lower constraint violation.

CCS Concepts: • **Information systems → Enterprise resource planning**.

Keywords: Cloud Computing, Container, Resource Scheduling, Reinforcement Learning

1 Introduction

Production clusters deploy a wide variety of *long-running applications* (LRAs) to provide real-time services in response to dynamic requests. Examples include stream processing [7, 9, 11, 78], storage services [8, 19, 22], machine learning [20, 26, 36, 43, 54, 69], and web services [24, 53]. These LRAs run a large number of instance replicas in *long-lived containers* which continue execution for hours to months [42, 44, 48, 73]. In comparison, conventional offline batch processing workloads (e.g., Spark and MapReduce jobs) run short-lived tasks that typically finish within minutes or shorter.

LRAs usually have stringent service-level objectives (SLOs) and demand a large number of resources in production clusters [42, 44, 48, 73]. Given the long duration of LRA containers, optimally placing container is critically important in attaining the best performance. However, this turns out to be particularly challenging as LRA containers have complex interference patterns. For instance, they often contend on shared resources such as memory bandwidth, I/O, and system cache. Co-locating contending containers in a physical machine can result in severe interferences. In addition, many LRA containers have I/O dependencies, where the output of one container can be the input for another in data processing

*Equal Contribution.

SoCC '21, November 1–4, 2021, Seattle, WA, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3486971

pipelines. Placing such dependent containers on separate machines is highly undesirable as it incurs significant communications across different machines.

To make the matter worse, container placements in production clusters are often subject to a number of *operation constraints*, specified by cluster operators to meet various deployment requirements. For example, a business-critical LRA may require its multiple instance replicas to run on different machines in order to provide high availability against node failures; when launching a new cloud service, phased deployment is commonly used, where container placements are confined to a small set of machines in the initial phase before committed in the entire cluster. Specifically, operation constraints can be categorized as *hard* or *soft* constraints, where hard constraints must be strictly enforced, while soft constraints can be violated within a pre-defined threshold at the expense of performance degradation, e.g., reduced fault tolerance.

Given these challenges, existing schedulers, either *rule-based* [2, 3, 17, 42, 71, 72] or *learning-based* [73], fall short in concerning LRA container scheduling under operation constraints. Rule-based schedulers rely on experts to summarize the sophisticated interference between containers, in particular in terms of *affinity* and *anti-affinity*, and explicitly express them as placement rules in scheduling. While this seems natural to support operation constraints, the fundamental limitation is that the constraints are represented only in a qualitative manner with manual specification. This is inaccurate and does not capture the quantitative effect on cluster performance [73]. Learning-based schedulers, on the other hand, by utilizing reinforcement learning techniques, can *automatically* place LRA containers. These schedulers learn to schedule LRAs from past workload logs or offline profiling with the intelligence in characterizing the complex interference between containers, however, none of the existing learning-based schedulers take into account the operation constraints.

Motivated by the objective of optimizing container performance subject to the operation constraints, in this paper we present George[1], a novel learning-based LRA scheduler. The core of George runs a tailored constrained policy optimization algorithm, which intelligently captures the interference between containers, complies with the strict operation constraints, and produces high-quality placement decisions efficiently. Our major contributions are summarized below.

Constrained scheduling policy. The core of George is a constrained scheduling policy. We propose a projection-based proximal policy optimization (PPPO) algorithm, which schedules LRA containers under the operation constraints intelligently. We seamlessly integrate PPPO with Integer Linear optimization techniques to handle the *hard* constraints.

[1]"George" is a colloquialism used unofficially to represent the autopilot system.

Compared with the traditional constrained policy optimization algorithms [27, 77] using RL techniques, our policy can satisfy all operation constraints and obtain high-quality LRA placement performance with significantly less training time.

Transfer learning. We design a transfer learning mechanism, which reduces the training time from hours to tens of minutes. The trace analysis [4, 15] shows that LRA scheduling events in clusters are similar in a long period of time, which implies that a well-trained LRA scheduler can be possibly reused. Therefore, upon a scheduling event, we opt to train a scheduler based on the results from prior scheduling events. We apply the transfer learning technique, which helps to reduce the training time drastically.

We have implemented George as a pluggable scheduling service in Docker Swarm [12]. We evaluate its performance in large-scale EC2 [6] clusters with over 700 machines. Compared with the state-of-the-art LRA schedulers, George satisfies the operation constraints and achieves substantially higher container performance. Compared with the state-of-the-art CPO algorithms [27], George reduces the RL model update time by up to 40%. With the transfer learning mechanism, the model training process is accelerated by 6×.

2 Background and Formulation

2.1 Long Running Application (LRA)

Cloud service providers, like Google [15] and Alibaba [4, 34], operate large production clusters and execute online cloud applications in supporting various interactive, latency-critical services such as stream processing [7, 9, 11, 78], interactive data analytics [59, 75], storage services [8, 19, 22], and machine learning applications [10, 20, 26, 36, 43, 54, 69]. These applications usually run in *long-lived containers*, in responding to dynamic queries in real-time. A study shows that in Microsoft clusters, the containers of online services typically run for hours to months [42]. Our communications with Alibaba Cloud also confirm this, where nearly 50% of Alibaba containers last over a day, and some of core applications run for several months.

LRAs occupy a large number of cluster resources to provide critical services with stringent SLOs [34, 42, 44, 48]. For instance, a recent trace from Alibaba of an 8-day period [4, 44] shows that 94.2% of the CPU cores on average are allocated for LRAs. In Microsoft, many large clusters are entirely dedicated to LRA workloads [42]. Therefore, scheduling LRAs plays a critical role in dictating both the overall performance and cluster utilization.

2.2 Placing LRA Containers

To provide high-available services, an LRA typically deploys multiple container replicas, each running a service instance. Over time, an LRA scheduler responds to continuous container launching requests in coping with the workload changes such as diurnal fluctuations and unexpected bursts. Given the dynamic arrivals of container launching requests, a scheduler needs to examine the current cluster state (i.e., the running containers on each machine) and determine the *placement* of each container by selecting a target physical machine.

Per-group Container Scheduling. To achieve better scheduling decisions in the *long run* and avoid myopic placements, a scheduler batches the received container launching requests into *groups*, each consisting of no more than T containers [42, 73]. The scheduler then makes placement decisions on a *per-group* basis. Once a container is deployed, it runs for an extended period of time, during which no preemption or migration is allowed.

Performance Interferences between Containers. Despite the continuous efforts on service isolation techniques [17, 79], the *interferences* between containers co-located on a physical machine are inevitable from two aspects: *On the negative side*, although CPU and memory of the co-located containers can be isolated properly, other resources—like network, disk I/O and cache, not managed by OS kernel—are not easily manageable, and containers competing for those resources may impair the performance of each other [42]. *On the positive side*, the container performance in term of throughput can be substantially improved if the data needed can be provisioned locally in co-locating containers—rather than through remote requests. Previous studies [42, 73] show that different container placement decisions cause up to 40% throughput variations.

Operation Constraints. Placing containers in *production clusters* is subject to a set of *operation constraints* such as hardware requirements, fault-tolerance, and incremental deployment [4, 15]. These constraints may require certain specific container placement in order to ensure steady and healthy cluster operations. Noticing, however, such constraints, often manually specified, are *independent* of container interferences. We summarize the commonly supported operation constraints as follows.

First, LRA containers should be placed to machines that match specific *hardware requirements* (e.g., GPU version, number of CPUs, minimum memory, available public IP addresses [15]), without which a container cannot run. *Second*, the container replicas launched by an LRA are typically placed on different physical machines, racks and zones for high-availability and disaster-avoidance. For example, in Alibaba clusters [4, 44], each online service has two or more container replicas, and these replicas are required to be scheduled on *different physical machines*, referred as *deployment spreading*. Consequently, a single machine failure will not crash any service. *Third,* placing new-version containers on specific machines for incremental updates. Online services are routinely updated to newer versions, which are usually deployed incrementally, i.e., deploying the new-version containers on a specific machine subset for A/B testing [1].

Hard and Soft Operation Constraints. In production clusters [4, 15], these operation constraints are *categorically* represented by two classes: *hard constraints* must be *strictly* enforced, and *soft constraints* should be guaranteed within a pre-defined violation threshold (e.g., 5%). On one hand, the constraints such as hardware requirement are naturally hard constraints without which applications cannot run, while others like the incremental deployment are soft constraints with certain violation tolerance. On the other hand, satisfying all operation constraints are costly and may not even be feasible in production clusters. From our conversations with Alibaba Cloud, out of thousands of LRAs running in clusters, only a handful of core LRA services are scheduled with 100% operation constraint guarantee, while others are running under bounded violation rates.

2.3 Placement Formulation

Based on the discussion above, an LRA scheduler should be aware of the inter-container interferences for better container performance, while categorically satisfying the operation constraints. We, therefore, formulate the LRA container placement as a *constrained combinatorial optimization problem* as follows.

Formulation. Formally, given an N-node cluster with node $n \in \{1 \dots N\}$, and a group of T containers $\{c_1, c_2, c_3, \dots, c_T\}$ to be placed, the scheduler determines the hosting machines for these containers, and outputs the placement decision $\mathcal{A} = \{a_1, a_2, a_3, \dots, a_T\}$, where $a_t \in \{1 \dots N\}$ is the node to host c_t. The objective is to maximize the container performance, subject to satisfying hard constraints and bounding the violation of soft constraints, i.e.,

$$\begin{aligned} \text{maximize} \quad & P(\mathcal{A}), \\ \text{s.t.} \quad & C^s(\mathcal{A}) \le h^s, \\ & C^h(\mathcal{A}) = 0. \end{aligned} \tag{1}$$

Here $P(\mathcal{A})$ measures the container performance under the placement $\mathcal{A}$, e.g., the average container service throughput. C^s and C^h measure the constraint violations for the soft and hard operation constraints, respectively. In particular, for hard constraints, the violation is guaranteed to be 0, and a violation threshold h^s is pre-defined for soft constraints.

2.4 Challenges

Solving the above constrained combinatorial optimization problem in Eq. (1) is technically challenging in several aspects.

Maximizing Container Performance Subject to Operation Constraints. Incorporating the operation constraints in LRA scheduling formulates a constrained optimization problem, which is fundamentally different from the previous formulations [73]. The constrained optimization problem itself is hard to solve [27, 77], as the performance optimality and the operation constraints are independent. Specifically, the operation constraints are usually set manually with no relevance to the container performance. Satisfying these constraints reduces the container placement decision space, which negatively affects the container performance.

Categorically Support Hard and Soft Operation Constraints. Categorically satisfying the hard and soft operation constraints further complicates the problem, as they demand different designs. The hard operation constraints have the highest priority and need to be strictly enforced, even at the expense of sacrificing the container performance. This also implies that a statistically low bound on hard constraint violation, like those in [27, 77], is not acceptable. In contrast, the pre-defined violation rate of soft constraints can be properly exploited to benefit the container performance.

Scalability. In large-scale clusters where tens of thousands of LRA containers run on thousands of machines, making *timely* placement decisions are critically important. Noticing that placing containers in a per-group manner shown in Eq. (1) requires a global view involving all containers, which is essentially a *combinatorial optimization problem*. The complexity grows exponentially with the increases of the cluster size and the number of containers in a group [42, 46]. The operation constraints further complicate this problem and significantly increase the scheduling latency. In Cloudera and Yahoo cluster traces, the scheduling latency is more than doubled [32, 33, 68], and in Google traces, the latency increases up to six times [62], because of the operation constraints.

3 Existing LRA Schedulers

In this section, we first outline the existing rule-based and learning-based LRA scheduling algorithms and discuss how they fail in optimizing performance or capturing the operation constraints. We then present the recent advances in machine learning techniques that explicitly consider constraints during policy learning, and we illustrate why they cannot be used in our LRA scheduling formulation.

3.1 Rule-based LRA Schedulers

Existing rule-based LRA schedulers use various *placement rules* to capture the complex interferences between LRA containers [2, 3, 17, 42, 71, 72]. The typical placement rules considered in cluster management systems [12, 17, 71] include *affinity*, which co-locates I/O-dependent containers to provide data locality or places containers to satisfy machines attributes, and *anti-affinity*, which schedules contending containers on separate machines to avoid resource interference or spreads container replicas for disaster avoidance. The operation constraints can also be *naturally* represented as placement rules. These rules are usually specified through scheduler-provided APIs [17, 42, 71]. The schedulers make container placement decisions to satisfy *as many rules as possible*, either with simple greedy strategies [38, 39] or by solving a constrained combinatorial optimization problem [42, 49, 56, 57, 70, 76].

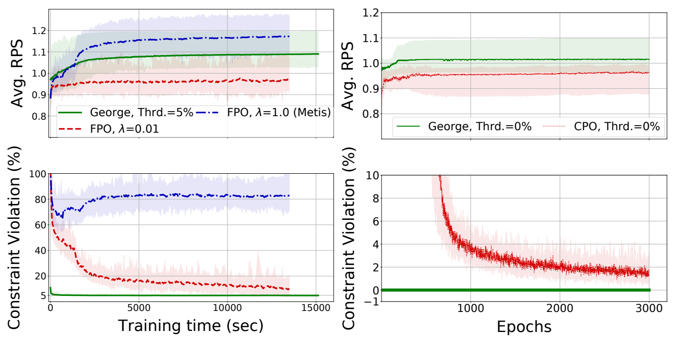
However, such rule-based LRA scheduling can be highly inefficient [73]. First, it requires manual summarization of placement rules, which is inefficient to capture inter-container interferences, especially in large-scale production clusters. Second, placement rules only provide qualitative scheduling guidelines, and cannot quantify the actual performance impact (e.g., to what degree can the throughput be compromised under certain constraint violations). In this case, when a scheduler cannot satisfy all the rules, it may well end up selecting some with negative impacts on the container performance. Third, complex placement constraints often lead to an extremely complex optimization formulation in large clusters. As we will see in § 6.3, when scheduling containers in a large cluster with 729 nodes, state-of-the-art rule-based LRA scheduler [42] spends more than 16 hours to search a feasible yet non-optimal solution, 16× longer than our proposed scheme.

3.2 Learning-based LRA schedulers

Another paradigm of LRA scheduling takes advantage of machine learning techniques in decision-making. The learning-based LRA scheduler [73] typically applies Reinforcement Learning (RL) techniques [29, 40, 64] to automatically learn the container placement decisions. More specifically, an RL agent learns to place containers by encoding the scheduling policy into a neural network and training it with trial-and-error experiments, in which it places LRA containers and iteratively refines the policy based on feedback. The state-of-the-art RL-based LRA scheduler, Metis [73] utilizes a novel REINFORCE algorithm [64], and decomposes the container placement into a series of sub-problems with hierarchical Reinforcement Learning (HRL) [29, 40] to achieve scalability.

RL-based solutions seem to be more promising compared with rule-based alternatives, as they eliminate the need to manually specify complex and inaccurate placement rules, and offer concrete *quantitative* scheduling criteria that *directly* help to improve the container performance in an *end-to-end* manner. In addition, by encoding the policy into the neural network, the optimal placement decision can be derived *timely* through model inference.

However, existing RL-based LRA schedulers [73] concern only container performance optimization, but do not consider operation constraints. Incorporating operation constraints results in a *constrained combinatorial optimization* formulation as shown in Eq. (1). This is more complex than the optimization formulation in [73]. First, solving a constrained optimization problem, especially satisfying hard constraints is challenging for RL, as we will discuss in § 3.3. Second, it often takes an excessively long training time to



(a) George vs. RL-based FPO. **(b)** George vs. RL-based CPO [27].

Figure 1. Inefficiency of existing constraint-aware RL algorithms. We adopt normalized RPS (requests per second) as the performance metric [39, 42].

solve the constrained combinatorial optimization problem when applying the existing RL algorithms [63].

3.3 Constraint-Aware Reinforcement Learning

We next discuss the difficulties for an RL agent in solving a *constrained combinatorial optimization* problem shown in Eq. (1), which motivates us to design a new RL algorithm.

Fixed Penalty Optimization (FPO). A naive way to incorporate constraints in an RL formulation is to combine the performance-oriented objective J_R and constraint violation J_C into the reward signal by a *weighted sum*, i.e.,

$$R_t = \lambda J_R - (1-\lambda) J_C, \tag{2}$$

where λ is a tunable knob that assigns different weights to the two objectives.

However, in FPO, the RL agent's behavior is highly *sensitive* to the hyper-parameter λ, as illustrated in Fig. 1a. A larger λ (e.g., $\lambda = 1$) leads to excessive constraint violations (the blue, dashed-dotted curve), while a smaller λ (e.g., $\lambda = 0.01$) leads to poor container performances (the red, dashed curve). It indicates that FPO requires painstaking hyperparameter tuning and is not a justified solution.

Constrained Policy Optimization (CPO). More sophisticated constraint-aware RL algorithms [27, 67, 77] are recently proposed in the area of robotics to prevent robots from dangerous actions. State-of-the-art algorithms such as CPO [27] and PCPO [77] explicitly formulate a constrained optimization problem. Through principled approximation and primal-dual optimization, the constrained optimization can be solved by iterative model training. Theoretical analysis proves these RL algorithms can simultaneously maximize the RL reward and approximately satisfy constraints.

However, we find these solutions cannot be directly used in our LRA scheduling formulation, due to the following two reasons. *First*, these algorithms do not guarantee *hard* constraints. As a matter of fact, *probability-based* RL algorithms cannot guarantee hard constraints. A probability-based RL agent makes decisions by sampling actions with different probabilities. Hence, any actions, including those violating hard constraints, are possible to be triggered. CPO and PCPO [27, 67, 77] only support soft constraints by limiting the *mathematical expectation* of the violations within a small but non-zero threshold. To further confirm this, we test CPO [27] with only hard constraints and set the threshold as 0. Experimental results reveal that hard constraint violations cannot be completely eliminated, as shown in Fig. 1b.

Second, these algorithms require complex second-order derivative calculation in the implementation [27, 60, 77], leading to an *unacceptable time complexity* when updating neural network parameters. More specifically, these algorithms use the trust region policy optimization method [60] to constrain an RL agent's behaviors, which relies on calculating the KL-divergence [27, 60]—the similarity between the current and the updated policies. The computational complexity grows rapidly when the neural network gets deeper [27, 60, 67, 77].

4 George Design

Given the inefficiencies in the existing schedulers, we now propose George, an intelligent RL-based LRA scheduler that enables scalable learning under the operation constraints in large clusters. We next present the key designs in George that target to solve the two problems: (1) How to categorically support the hard and soft operation constraints during learning process? (2) How to provide scalability with timely decision-making? We illustrate how George trains a learning-based scheduler with the key designs in Algorithm 1.

4.1 Preliminaries

RL Workflow. Recall in § 2 that a scheduler schedules a group of T containers at their arrivals. We treat each group scheduling as an *episode* consisting of T steps, where in each step t, only one container c_t is placed onto a machine a_t. Each placement decision is made by George's RL agents, which encodes the scheduling policy into a neural network with parameters θ, known as policy network π_θ. After all the T containers have been placed, the scheduler evaluates the outcome of the placement $\mathcal{A} = \{a_1, a_2, \ldots, a_T\}$ with two parts: $P(\mathcal{A})$—the container performance, and $C^h(\mathcal{A})$, $C^s(\mathcal{A})$—the violations of the hard and soft operation constraints, respectively. Based on these feedbacks, George updates its policy π_θ iteratively.

State and Action. At each step t in an episode, assume c_t is the next container to be scheduled. To embed container c_t, we encode it into a *one-hot vector* $\boldsymbol{e} = \langle e_1, \ldots, e_M \rangle$, where each element e_i is 1 if container c_t belongs to application i else 0. To represent the current cluster state, we first define the *state of a machine n* as the vector $\boldsymbol{v}_n = \langle v_1, \ldots, v_M \rangle$, where v_i is the number of containers that run for application i. The concatenation of *all machines' states* and the one-hot vector $\boldsymbol{e}$ defines the *cluster state* $\boldsymbol{s}_t$. The agent chooses to perform *action* a_t in state $\boldsymbol{s}_t$, which schedules container c_t to machine a_t and transits the system to a new state.

Reward and Cost. In each episode, the agent evaluates the container performance and examines the constraint violations of the group placement in the final step T after *all* T containers are scheduled. We denote the container performance objective as the *reward* function $J_R(\pi_\theta)$, which can be any performance measurements such as the average container throughput, SLO satisfaction rate, cluster utilizations, or their combinations. We represent the constraint violation as the *cost* function $J_C(\pi_\theta) = J_C^h(\pi_\theta) + J_C^s(\pi_\theta)$, a summed violation rates of soft and hard constraints. In RL training, it is common to discount the future reward and cost with a discount factor $\gamma \in [0, 1)$ [51, 60, 61, 73, 77, 77]. In policy optimization, we consider the reward advantage function $A_R^{\pi_\theta}(\boldsymbol{s_t}, a_t)$, which measures the quality of an action a_t at the state $\boldsymbol{s_t}$ and is defined as the difference between the value of reward and its expectation.

Designs Inherited from Metis [73]. Inspired by Metis [73], George adopts three mechanisms to enable scalable learning in large-scale clusters.

1) ***Cluster Environment Simulator.*** Similar to [28, 51, 73], we develop a high-fidelity cluster environment simulator that can faithfully predict containers' performance immediately after a placement. This enables the RL policy to be trained in a very efficient manner, without the need to interact with real clusters. Previous work [73] reveals that collecting only 20% of the container co-location samples is sufficient for a Random Forests [30] regressor to accurately predict the container performance (e.g., container throughput) under any possible co-locations.

2) ***In-place Model Training.*** George trains a dedicated RL model upon the arrival of each new group of containers. Note that the possible combinations in a container group is large. For example, a small group of 30 containers launched by seven LRAs requires the scheduler to handle over one million possible container combinations in input. Due to such *highly-variant* possible input, *offline trained* RL models inevitably result in poor scheduling performance [52].

3) ***Hierarchical Cluster Decompositions.*** George incorporates the hierarchical RL (HRL) training framework from Metis [73] for manageable RL state and action space. George *recursively* divides the original large cluster into K equal-sized *sub-clusters* at multiple layers. This decomposition establishes a *decision tree*, where each tree node corresponds to a sub-task that selects one smaller sub-cluster at the next layer out of the K candidates, as shown in Fig. 2. Following such as decision tree, George places each container by recursively making several K-choose-1 decisions, each is independently modeled as an RL problem with its own state, action, reward and cost. With such an HRL design, scaling the

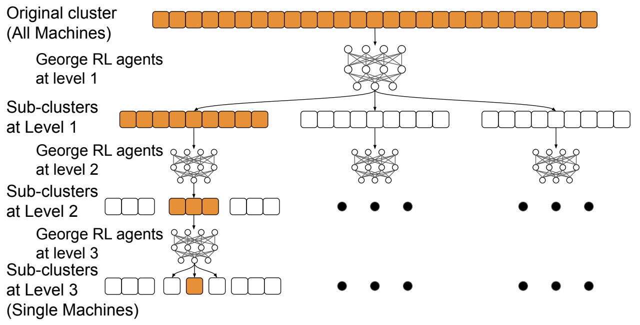

Original cluster (All Machines)
George RL agents at level 1
Sub-clusters at Level 1
George RL agents at level 2
Sub-clusters at Level 2
George RL agents at level 3
Sub-clusters at Level 3 (Single Machines)

Figure 2. An illustration of hierarchical cluster decompositions from Metis [73]. Following this design, George recursively divides the original cluster into K equal-sized subclusters at multiple levels, transforming the original problem into a sequence of K-choose-1 subproblems.

cluster with more machines only leads to a deeper decision tree and more sub-tasks, but each sub-task still maintains manageable state/action space.

4.2 Satisfy Hard Constraints with ILP-Based Filter

Insights. As discussed in Sec 3, RL algorithms cannot satisfy hard operation constraints. To address this problem, we divide the decision-making process when placing each container into two stages: At the first stage, we *filter* the original action space—the entire cluster with all machine candidates—to exclude those actions that will violate hard constraints. In another word, after the filtering, the action space is in a "safe action space", in which any actions cannot violate hard constraints. We next proceed to the second stage, in which the RL agent determines the placement within this "safe action space" by focusing on maximizing container performances subject to a set of soft constraints.

Difficulty in Filtering the Action Space. George filters the original action space by identifying and excluding those "unsafe" actions. In simple cases where the "unsafe" actions are independent and easily identified, we can directly exclude those actions. However, for sequential actions (i.e. placing a group of containers), it becomes difficult to *myopically* ensure if an action will eventually violate hard constraints. In specific, when filtering the action space, a scheduler needs to have a *global view* of all containers in the group and all hard constraints, filtering "unsafe" actions *in the long term.*

Filtering the Action Space with ILP. We propose that the filtering decisions can be derived by integer linear programming (ILP) techniques [32, 42]. Formally, given a T-container group $\{c_1, c_2, \ldots, c_T\}$, assume c_t is the next to place. The original action space is the entire cluster, i.e., $a_t \in \{1 \ldots N\}$, where N is the cluster size. For each node $n \in \{1 \ldots N\}$, suppose it is the placement decision for c_t (i.e., $a_t = n$), George verifies if the remaining containers $\{c_{t+1}, c_{t+2}, \ldots, c_T\}$ can be placed without violating hard constraints, i.e., to check if the following integer programming problem has a *feasible solution*:

$$C^h(\mathcal{A} = \{a_{t+1}, a_{t+2}, \ldots a_T | a_1, a_2, \ldots, a_t = n\}) = 0. \quad (3)$$

By checking each node n, we obtain a "safe actions", based on which George then makes the placement decision for container c_t using RL-based algorithms.

Benefits. Compared with rule-based schedulers [38, 42] and learning-based schedulers [73], the two-stage method seamlessly integrates the intelligence of RL algorithms and the capability of ILP in supporting hard constraints.

Algorithm 1 Training process in George

```
Input An $N$-node cluster; a group of $T$ containers; Policy $\pi_0$ from a pre-trained base model or random initialization;
Output An allocation of $\{a_1, a_2, a_3, \ldots, a_T\}$ of the $T$ containers;
    Initialize the environment, performance indicator $R$, and the threshold for constraint violations $h$.
    Set replay buffer $B$ and the best performance $R^*$ as empty.
    for epoch k=1,2,...K do
        Initialize the state $s = \{e, v_1, v_2, \ldots, v_N\}$.
        for t=1,2,...T do
            Check the action space with ILP-based filter
            Choose an action $a_t$ with policy $\pi_k(a_t \mid s_t)$.
            Execute the action and observe a new state $s_{t+1}$.
        Collect all performance indicators as reward $r = \sum_t R(c_t)$
        Collect all constraint violations as cost $c = \sum_t C(c_t)$
        if $r \geq R^*$    $c \leq h$ then
            Store experience $\{s_1, a_1, s_2, a_2, \ldots, s_T, a_T\}$ in B
            $R^* \leftarrow max(r, R*)$
        Use PPPO algorithm to update $\pi_k$
    Return the action $\{a_1, a_2, \ldots a_T\}$ of the experience with the highest reward $r = R^*$ in replay buffer $B$.
```

4.3 Efficient Policy Optimization with PPPO

To satisfy soft constraints, we next present a projection-based proximal policy optimization (PPPO) algorithm, a first-order constrained policy optimization technique. We show PPPO drastically reduces the computation complexity for policy updates, with guaranteed performance improvement and bounded constraint violation.

Insights. It turns out that the second-order derivative calculation, i.e. the computation of the inverse of Hessian matrix, in existing constraint-aware algorithms [27, 77] can be *approximated* by a first-order alternative, with much reduced complexity and bounded performance. To obtain such an approximation, we take advantage of the two-step strategy from [77], which decouples the constrained policy optimization into a reward improvement step that only concerns the performance-oriented goals and a projection step that projects the policy update to a "safe zone" that satisfies soft constraints.

Reward Improvement Step. The reward improvement step is designed to optimize performance-oriented goals. We perform an update to optimize the policy by maximizing a "surrogate" objective function as Eq. (4) [27, 60, 77], i.e.,

$$L(\theta) = \mathbb{E}[\frac{\pi_\theta\ (a_t \mid s_t)}{\pi_{\theta_{\text{old}}}\ (a_t \mid s_t)} A_R^{\pi_{\theta_{\text{old}}}}(s_t, a_t)]. \quad (4)$$

Intuitively, Eq. (4) suggests that the probability an action a_t at state s_t increases or decreases, if the advantage function of the pair (s_t, a_t) is positive or negative, as a positive or negative advantage function indicates the quality of the action a_t is above or below the average.

While updating the policy, existing algorithms [27, 67, 77] apply a trust region method [60] to limit the differences between π_θ and $\pi_{\theta_{\text{old}}}$ with a KL-divergence inequality $D_{KL}(\pi_\theta \parallel \pi_{\theta_{\text{old}}}) \leq \delta$. However, the calculation of KL divergence involves computing the inverse of its Hessian matrix [60], making its computation intractable especially for high-dimension policies, e.g., policy network. Inspired by [61], we replace KL-divergence constraint with a clipping function, $clip(input, 1-\epsilon, 1+\epsilon)$, which limits the input term in the interval $[1-\epsilon, 1+\epsilon]$. We therefore transform Eq. (4) as

$$L(\theta) = \mathbb{E}\left[\min\left(r_t(\theta) A_R^{\pi_{\theta_{\text{old}}}}, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) A_R^{\pi_{\theta_{\text{old}}}}\right)\right], \quad (5)$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$. Denote α the learning rate. Take the first order gradient of Eq. (5) with respect to θ. The update rule in the reward improvement step of the k^{th} epoch is

$$\theta_{k+\frac{1}{2}} = \theta_k + \alpha \sum_t \nabla_\theta L(\theta_k). \quad (6)$$

Projection Step. Considering that the updated scheduling policy $\pi_{k+\frac{1}{2}}$ may violate the operation constraints, we then perform the projection step to project it into a "safe" zone, in which $J_C(\pi) \leq h$. The projection step is performed by minimizing a distance measure D, between $\pi_{k+\frac{1}{2}}$ and π_k while controlling operation constraint violations:

$$\begin{aligned} \pi_{k+1} = \arg\min_\pi \quad & D\left(\pi, \pi_{k+\frac{1}{2}}\right) \\ \text{s.t. } & J_C(\pi) < h. \end{aligned} \quad (7)$$

The projection step guarantees that the constraint-satisfying policy π_{k+1} is close to $\pi_{k+\frac{1}{2}}$, subject to constraints. Let $\boldsymbol{a}$ be the first-order derivative of the cost function $J_C(\pi_\theta)$. By solving Eq. (7) with convex programming [77], we obtain the

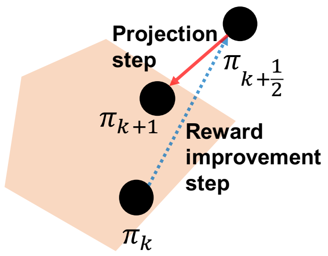


Figure 3. Update procedures for PPPO: Reward improvement and projection steps. The first step (blue dashed arrow) follows the reward improvement direction, leading to $\pi_{k+\frac{1}{2}}$. The second step (red solid arrow) projects the policy $\pi_{k+\frac{1}{2}}$ onto the "safe" zone (light orange), leading to policy π_{k+1}.

projection step which projects the policy update to the "safe" zone, i.e.,

$$\theta_{k+1} = \theta_{k+\frac{1}{2}} - \max\left(0, \frac{\boldsymbol{a}^T\left(\theta_{k+\frac{1}{2}} - \theta_k\right) + \max\left(0, J_C\left(\pi_{\theta_k}\right) - h\right)}{\boldsymbol{a}^T L^{-1} \boldsymbol{a}^T}\right) L^{-1} \boldsymbol{a}. \tag{8}$$

We derive Eq. (8) based on the Lagrangian of the Eq. (7) and Karush-Kuhn-Tucker (KKT) conditions. The detailed solution is included in [14]. Fig. 3 depicts the reward improvement and projection steps in PPPO. Compared with CPO and PCPO [27, 77], PPPO eliminates the complex calculation of KL-divergence and achieves faster model update. As we will show in § 6.2, PPPO accelerates the model updating time by by 40% on average without compromising the scheduling performance.

Performance Bounds. We analyze the worst-case performance of the proposed PPPO. The following two theorems provide a lower bound on reward improvement, and an upper bound on constraint violation for each policy update.

Theorem 1 (Lower bound on reward improvement)**.** *For any policies π', π, with $\epsilon^{\pi'} \doteq \max_s \left|\mathrm{E}_{a\sim\pi'}\left[A_R^{\pi}(s,a)\right]\right|$ and $\gamma \in [0,1)$, the following bound holds:*

$$J_R(\pi') - J_R(\pi) \geq -\frac{\sqrt{2}\gamma\epsilon^{\pi'}}{(1-\gamma)^2}\sqrt{\log(1+\epsilon)}. \tag{9}$$

Theorem 2 (Upper bound on constraint violation)**.** *For any policies π', π, with $\epsilon_C^{\pi'} \doteq \max_s \left|\mathrm{E}_{a\sim\pi'}\left[A_C^{\pi}(s,a)\right]\right|$ and $\gamma \in [0,1)$, the following bound holds:*

$$J_C(\pi') \leq h + \frac{\sqrt{2}\gamma\epsilon_C^{\pi'}}{(1-\gamma)^2}\sqrt{\log(1+\epsilon)}. \tag{10}$$

Theorem 1 provides a worst-case performance degradation guarantee which depends on the hyperparameter ϵ. ϵ captures the difference between the new and the old policies, which is often set below 0.2. Theorem 2 provides a performance guarantee for the satisfaction of constraints. Here, h is the pre-defined threshold on the percentage of constraint violation.

The general idea of their proofs follows two steps. First, we bound the differences in J and J_C between two arbitrary policies π', π in terms of the total variational divergence (TV-divergence) between their action distributions [27]. Then we relate the TV-divergence to the clipping functions in Eq. (5) and get final results step by step. The detailed derivation and mathematical analysis are included in [14].

4.4 Temporal Model Reuse: Transfer Learning

Motivation. Although our PPPO algorithm accelerates the RL agent learning process, the model training is still time-consuming which may not be practically feasible in making timely decisions, esp., for large clusters. As we will show in Fig. 7c, scheduling thousands of containers in a large cluster with ~700 machines requires more than 6 hours for the training convergence. Such a long scheduling latency may not be justifiable in real-world production cluster operations.

Insights. We note that training a tailored RL agent *from scratch* for each container scheduling event is unnecessary. This is because, although different scheduling events involve different container groups and different RL models, the comprehension of the problem and learned knowledge can be *temporally reused* across different scheduling events.

Transfer Learning (TL). George addresses the problem of rapid model training by leveraging transfer learning techniques in the domain of RL [31, 65, 66]. The basic idea is that experiences gained in learning to perform one task can help improve learning performance in related, but different tasks.

Instead of training an RL model from scratch, George's model training is initialized based on a pre-trained model, referred to as *base model*. The base model can be obtained either from a previous scheduling event or a general offline trained model. George inherits the neural network parameters of the input layers and hidden layers from the base model, while randomly initializing the parameters of the output layer. By inheriting these parameters, the knowledge learned previously is transferred and reused in a new scheduling event, which considerably accelerates the model training. As we will show in Fig. 7c, employing transfer learning speeds up the model training by 6× than training each model from scratch.

Transfer Learning vs. Offline-trained Model. Previous study [73] reveals that training a unified RL model offline and applying it for inference online for various container scheduling events may not perform well. In general, a container group produces a large number of combinations, and an LRA scheduler needs to handle a highly variant input—in our experiments with seven applications (Fig. 1), scheduling a group consisting of 30 containers requires the scheduler to handle over one million possible container combinations

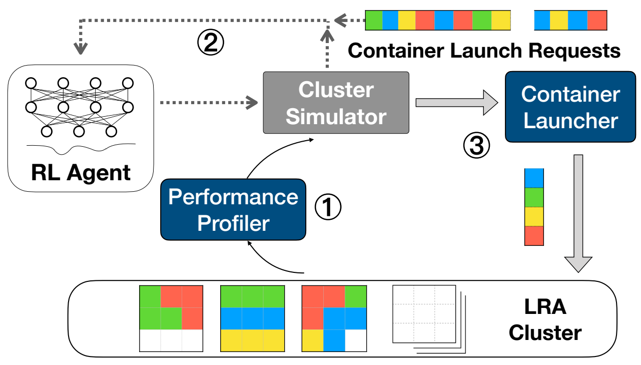


Figure 4. Docker Swarm Implementation of George.

as input. With a large cluster, offline training a scheduling policy for extremely variant workloads inevitably results in poor performance. In contrast, transfer learning techniques inherit the learned knowledge from previous scheduling events, while still having the opportunity to *refine* the model into a specific container group online. Even without considering the operation constraints, temporally reusing the RL models with transfer learning can also speed up the decision-making process for other RL-based LRA schedulers like Metis [73]. We believe this can be extended into other RL applications, especially for RL-based scheduling where the subsequent decision-making events share high-degree similarities.

5 Implementation

We implement George as a pluggable scheduling service in Docker Swarm [12], which is widely compatible with various container orchestrations [3, 17]. Fig. 4 describes the components in our RL-based scheduling algorithm[2].

Container Scheduling Component and Workflow. As shown in Fig. 4, given an LRA cluster, George first collects abundant container co-location data samples by an *Performance Profiler*, using direct benchmarking [18, 25] or trace analysis [4, 15, 45] (①). More specifically, it logs the machine-level container co-location vectors along with the observed RPS/latency of each resident container. The collected data is fed to a *Cluster Environment Simulator* (①). The Cluster Environment Simulator builds a `RandomForestRegressor` with scikit-learn [58] which takes an ensemble of 100 classifying decision trees with a maximum depth of 20. Based on the Cluster Environment Simulator, for each group of container launch requests, George initializes the RL model training, which is an iterative interaction process between the Cluster Environment Simulator and the *RL agent* (②). Once the model is converged, the optimized placement decision is produced, which is used by the *Container Launcher* to execute the actual container placement (③).

[2]We have open-sourced George on GitHub [13].

Parallel RL Model Inference with Ray [55] George batches episodes of agent-environment interactions and trains the policy network in a mini-batch manner. To speed up the batching process, we distribute the agent-environment interaction tasks to processes with Ray [55] and execute them in a synchronized parallel manner. This can substantially reduce the data sampling time in RL training.

ILP-based Filter George verifies the feasibility of the ILP problem using Z3 solver [37]. Z3 is a state-of-the-art theorem prover from Microsoft Research. It can be used to check the satisfiability of logical formulas over one or more theories. The set of formulas in the Z3 stack is *satisfiable* if there is an interpretation that makes all asserted formulas true. While placing LRA containers, the ILP-based filter helps George avoid hard constraint violations.

PPPO Algorithm The PPPO algorithm in the RL agent is written in Python code. We encode its RL policy into a 2-layer policy neural network and implement it with TensorFlow [26]. As shown in Algorithm 1, PPPO follows the ILP-based filter to update scheduling policy.

6 Evaluation

In this section, we evaluate George on EC2 clusters using container workloads of seven real applications covering machine learning, stream processing, I/O and storage services. Our evaluations focus on addressing the following three questions: (1) How does George perform compared to the rule-based LRA schedulers? (§ 6.2) (2) Can George schedule LRAs in large clusters with short model training time? (§ 6.3) (3) Can George support different scheduling objectives? (Sec. 6.4)

Our evaluations aiming to answer the above questions are conducted in three cluster scales. The first is a small-sized cluster with 27 nodes. We examine and compare the scheduling performance and constraint violations between George and baseline schedulers by focusing on the first and third questions. The second is a large-sized cluster with 729 nodes, in which we evaluate the scalability of George, and illustrate the benefits of transfer learning in addressing the second question. Lastly, we run George at a micro-benchmark level in the third tiny-sized cluster with only 9 nodes and present some of the detailed scheduling behaviors.

6.1 Methodology

Cluster Setup. We use the small-sized cluster with 27 nodes in §6.2 and §6.4, the large cluster with 729 nodes in §6.3 and the tiny-sized cluster with 9 nodes in §6.5. All the containers are running with the same resource demands of 2vCPU and 8GB memory. For simplicity, we assume a homogeneous cluster with identical machines [3]. For each machine, we use an `m5.4xlarge` AWS EC2 instance [5] with 16 vCPU and 64 GB RAM, hosting at most 8 containers simultaneously.

Workloads. We have implemented seven open-source LRAs:

- Redis [22]: a stand-alone Redis server instance with `redis-benchmark`[23] requests.
- MXNet Model Server (MMS) [20]: executing DNN models with MXNet [10] for image classification.
- Image Super Resolution (ISR) [41]: super-scaling low-resolution images by executing DNN.
- File Checksum (CKM) [16]: loading files from disks, hashing and verifying the checksums.
- Yahoo! Cloud Serving Benchmark workload A & B (YCSB-A and YCSB-B) [35]: read-write-balanced (A) or read-heavy (B) workloads [35].
- Video Scene Detection (ScD) [21]: differentiate colors between video frames to detect scene changes.

Metrics. We focus on both the container performance (RPS) and constraint satisfaction.

1) ***Container performance metric:*** We adopt *RPS* (requests per second) as the performance metric [42]. Particularly, we normalize values by a container's *stand-alone* RPS.

2) ***Operation constraint satisfaction metric:*** We measure *constraint violation*, the percentage of containers that violate the operation constraints.

LRA Interactions. Our LRAs have sophisticated performance interactions, here we focus on the following two:

1) ***Affinity:*** Owing to data dependencies, both MMS and ISR containers benefit from co-located Redis instances by caching input images and accessing them locally. However, such benefit is *one-way* as co-location harms the RPS of the Redis instance, due to the memory bandwidth competition.

2) ***Anti-affinity:*** Co-locating YCSB-A/B, Redis, CKM, and ScD containers reduces RPS as they contend on CPU cache and memory bandwidth. In particular, CKM and ScD need large cache space close to CPUs and high memory bandwidth for fast file read, whereas Redis and Memcached in YCSB-A/B are sensitive to these resources.

Operation Constraints We set four operation constraints based on production traces [4, 34, 48].

- *Node Capacity (hard constraints):* The number of containers placed to a machine should not exceed the node capacity, which is 8 in our experiment.
- *Deployment Spreading (hard constraints):* Each LRA can launch at most one container on a machine for load-balance.
- *Incremental deployment (soft constraints):* We use one LRA, ISR, to mimic the new-version deployment scenario. The containers are limited to be placed on a specific machine subset (50% of the machines).
- *Hardware affinity (soft constraints):* To mimic the hardware requirement, we assume one LRA, CKM, requires a particular kernel version and can only be deployed on a machine subset (30% of the machines).

Baselines We evaluate George against the state-of-the-art rule-based and RL-based schedulers:

1) ***Medea*** [42]. Similar to [73], we exhaustively profile the container co-location performance, and specify affinity/anti-affinity performance constraints. We fix the weight of performance constraints as 1, while associating the operation constraints with various weight values β for $\beta \in \{1, 10\}$. Based on these constraints, Medea solves an ILP problem using the branch-and-bound heuristic [47].

2) ***Fixed Penalty Optimization (FPO).*** We modify the RL-based LRA scheduler, Metis [73] to incorporate the operation constraints by revising the reward function as the combination of two parts: rewarding high container RPS and punishing high constraint violation: $\lambda J_R(\pi_\theta) - (1-\lambda) J_C(\pi_\theta)$ for $\lambda \in \{0.01, 1.0\}$. A smaller λ implies a higher preference to satisfy the operation constraints.

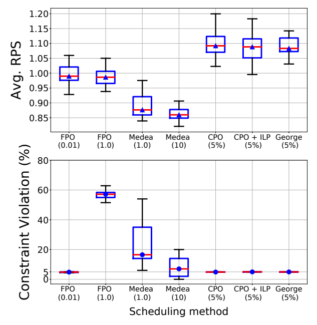
3) ***Constrained Policy Optimization (CPO).*** We also modify Metis [73] by replacing its Policy Gradient algorithm with CPO [27], the state-of-the-art constraint-aware RL method. We set the constraint violation threshold as 5%.

6.2 Schedule LRAs with PPPO and ILP-Based Filter

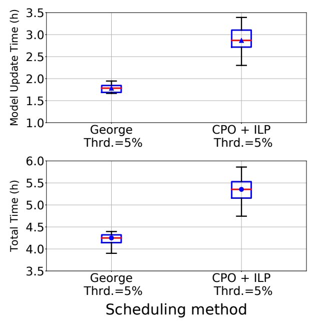
We first compare George with baseline schedulers in terms of container RPS, constraint violations and model training time. To identify and isolate the contributions from PPPO and ILP-based filter, we do not incorporate transfer learning in this part. The experiments are carried out in a small-sized cluster with 27 nodes. For each evaluated case, we randomly generate 30 container groups for statistical analysis. For George and CPO, we set the violation threshold of the soft constraints as 5%.

The End-to-end Scheduling Comparison Fig. 5a compares the average RPS and constraint violations under different schedulers. As we can see, the normalized RPS under George is consistently higher than 1, meaning the container throughput is higher than its stand-alone RPS. We attribute this to George's awareness of the inter-container performance interferences. Moreover, the soft constraint violation is steadily controlled to under 5%, the pre-defined threshold, while the hard constraints are strictly enforced. In contrast, both Medea and FPO are sensitive to the weighted knob: a higher value of λ and a lower value of β lead to higher container RPS, but more constraint violations. On the other hand, a lower value of λ and a higher value of β inevitably hurt

[3]Our approach can easily incorporate heterogeneous clusters and different container resource demands by embedding machine/container properties into the RL state.



(a) RPS and Constraint Violation.



(b) Scheduling Latency.

Figure 5. George vs baselines in small clusters with 27 nodes.

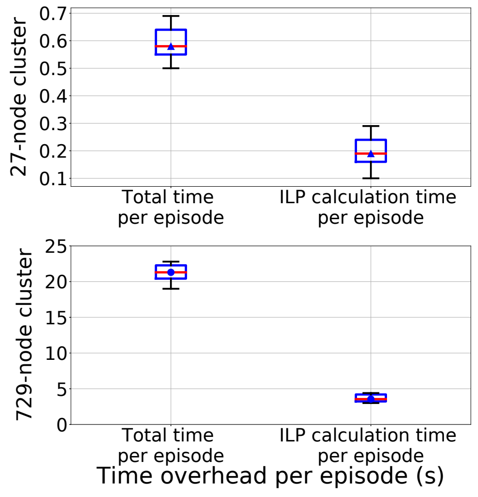


Figure 6. Additional calculation time of ILP-based filter.

the container RPS. In particular, neither Medea nor FPO can enforce constraint violations, while George limits it under 5%. At the same time, George provides 16% and 26% higher RPS than FPO and Medea, respectively.

Speed Up the Model Training with PPPO. Fig. 5b compares end-to-end scheduling latency (including model training and inference time) and the model training time between CPO [27] and PPPO. Both algorithms are executed within a filtered "safe action space" using our ILP-based filter. As we can see, our PPPO algorithm accelerates the model training time by 40% on average without compromising the scheduling performance. The end-to-end scheduling latency is also reduced by 20% on average.

Satisfy Hard Constraints with ILP-Based Filter. George fundamentally differs from all existing schedulers by categorically satisfying the *hard* and *soft* operation constraints. To concretely illustrate this, we generate 10 container groups each with 100 containers for George to schedule. We configure all the operation constraints are *hard constraints*, i.e., the violation threshold is 0, and compare the performance between George and CPO [27] algorithm. Fig. 1b illustrates the learning curve of George and CPO [27]. As we can see in the red dotted line, CPO by no means controls the constraint violation to zero although the pre-defined violation threshold is set as 0. In contrast, George's ILP design can ensure no hard constraint is violated during the entire training process.

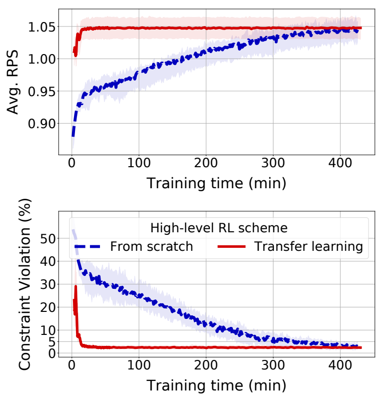
Additional Calculation Time of ILP-Based Filter. While using ILP techniques can strictly satisfy the hard operation constraints, it causes additional calculation time to check if the optimization problem shown in Eq. (3) is feasible. Fig. 6 quantifies this time consumption when scheduling containers in a small-size cluster with 27 nodes and a large cluster with 729 nodes. As we can see, when scheduling in a small-sized cluster with 27 nodes, the ILP calculation time per episode is less than 0.3s (~30%). For a large cluster with 729 nodes, this time is still less than 5s per episode (~20%). We believe such a calculation time is well justified in large clusters.

6.3 Accelerate RL Training with Transfer Learning

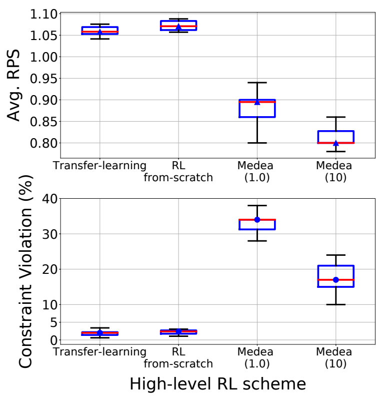
To better examine the effect of transfer learning technique, we run George to schedule 2000 containers in a *large* cluster of 729 nodes. We generate 30 container groups for statistical analysis, and the violation threshold of soft constraints is also set at 5%.

Transfer Learning vs. Training from Scratch. Fig. 7a compares the training process of two RL schemes under George: training from scratch for each scheduling event, and refining the training models with transfer learning based on a pre-trained general model. It is clear that training from scratch suffers from a long training time, i.e., spending more than 6 hours before model convergence, as shown with the blue dashed curve. This long latency is primarily caused by the computation complexity. In particular, scheduling containers in such a large cluster requires an exponentially large solution space, within which searching for a local optimal solution is practically difficult. Employing transfer learning technique can temporally reuse the previous model, where the model training is based on a pre-trained general model. Compared with training an RL model from scratch, this significantly accelerates the learning process, as shown in the red solid curve. Fig. 7a and Fig. 7b further demonstrate that, reusing the model provides comparable container RPS and constraint violation as training from scratch.

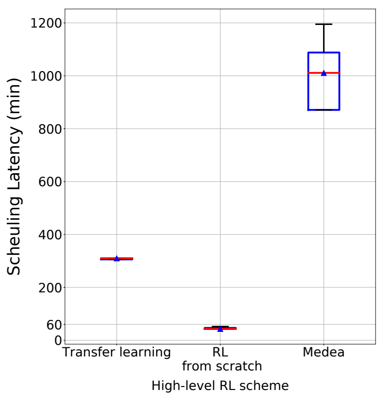
George vs. Medea. Medea [42], as discussed in [73], can not make timely decisions when the cluster size increases, due to the hardness in solving the ILP heuristics. Fig. 7c confirms that, Medea spends more than 16 hours to reach a sub-optimal solution, 16× higher than George. In addition, the scheduling performance under Medea suffers up to 23%



(a) Training Process.



(b) RPS and Constraint Violation.



(c) Scheduling Latency.

Figure 7. Transfer learning speeds up George's model training in a **large** cluster with 729 nodes.

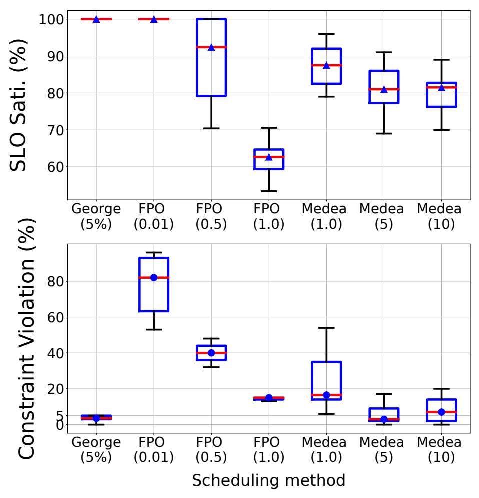


Figure 8. George supports maximizing SLO stratification.

lower container RPS or up to 8.5x higher constraint violations than George.

6.4 Various Scheduling Objectives

Motivated by George's intelligent scheduling mechanism described earlier, we next demonstrate that George is a general-purpose scheduler that can support various objectives beyond maximizing the average container RPS. The experiments below are conducted in a 27-node cluster.

Case Study: Maximizing Container RPS SLO Satisfactions. LRAs running in production cluster often have stringent SLO (Service-Level objective). Apart from maximizing the average container RPS as shown in Fig. 1, George can directly maximize the SLO satisfactions (i.e., the percentage of containers whose RPS meet a pre-defined requirement) while still satisfying the operation constraints. For illustration, we set the RPS SLO as 0.8, meaning that this requires the RPS of each container to be higher than 80% of the RPS when the container running alone on the machine.

Fig. 8 compares the SLO satisfaction rates under different schedulers. Compared with baselines, George consistently achieves both high SLO satisfactions and under-controlled constraint violations. In particular, under George, all the containers can meet the SLO requirement (RPS > 0.8) with less than 5% violation of the operation constraints. To achieve the same SLO satisfaction, we configure $\lambda = 0.1$ for Metis [73] with FPO, which however leads to more than 80% violation of the constraints. A higher value of λ (1.0), on the other hand, results in more than 30% SLO satisfaction violations. Medea [42] also encounter similar problem as shown in Fig. 8.

6.5 A Close Look at George's Behavior

We next run George in a tiny-sized cluster and examine its placement decisions at a micro-benchmark level in comparison with Metis [73]. We consider a cluster with 9 nodes and a group of 67 containers.

Operation Constraints vs. Performance Interferences. For simplicity, we only set one soft constraint: {Redis, MMS} should *not* be co-located on the same machine, and one hard constraint: each node has a capacity of 8. We set the threshold of constraint violation as 6%[4]. Note that, {Redis, MMS}, and {Redis, ISR} should be co-located on the same nodes for better performance, owing to the data dependencies, as discussed in § 6.1. However, our soft constraint here requires {Redis, MMS} to be separated.

Results. George strives for a balance in such a conflicting scenario, in which it places all ISR containers as co-located with Redis containers due to the affinity requirement. For MMS, George exploits the violation threshold (i.e. 6%) to maximize container performance. Only one Redis container and three MMS containers are co-located due to the affinity

[4]Setting 6% as the threshold yields only one optimal solution in this setting.

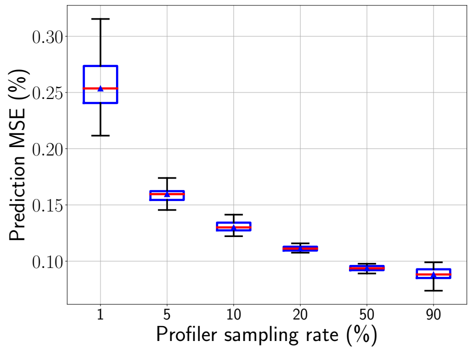


Figure 9. Prediction accuracy of the Cluster Environment Simulator (§5) with varying data sampling rates.

requirement, leading to an acceptable operation constraint violation, i.e. $5.97\% < 6\%$. The remaining containers on other nodes all satisfy the operation constraints. Metis satisfies the affinity between LRA containers, in which all containers of MMS and ISR are co-located with those of Redis. However, it does not address the operation constraints, leading to a 100.0% constraint violation.

6.6 Cluster Environment Simulator

As RL training critically relies on the prediction given by the Cluster Environment Simulator (§5), we validate its accuracy with different training samples. In particular, we profile the resultant container RPS in all 6435 possible container co-location options of the seven LRAs on one machine. We randomly sample the training sets covering 1%–90% of the profiled results—the remaining data is used as the test set. We train the predictor of the simulator with a multivariate Random Forests regression model (an ensemble of 100 decision trees with a maximum depth of 20), and evaluate its accuracy in terms of mean square error (MSE) over the test set. We find collecting only **20%** co-location samples is sufficient to suppress MSE below 0.15%. As generally recognized, Random Forest regression model produces good performance even when dealing with small sample sizes and high-dimensional feature spaces [30].

7 Discussion

Handling Workload Changes George's RL model depends on the representative training set, and one concern would be how it performs in handling new LRAs. From our conversations with Alibaba Cloud, the set of core LRA services is *stable* over time, where the containers are performance-critical and require dedicated placements. When incorporating a new LRA set, one may need to retrain the RL model before scheduling. Our Cluster Simulator (§5) enables the training to be conducted in a simulated environment quickly. Though, George's current RL design can only afford to handle dozens of core applications, we notice from production traces [4, 50] that only 21 out of 9k LRAs have more than 200 containers co-existing in the cluster.

Handling Varying Request Loads LRA services exhibit diverse request patterns such as diurnal changes and workload spikes [74]. To measure the container performance inferences, we stress-tested each LRA by configuring a constant and high load, which simulates the most challenging scenario in rush hours where sustained high loads stretch the processing capabilities of all LRA containers, creating the most severe interferences. We leave the evaluation under different workloads in our future work.

Handling Heterogeneous Containers Currently, George only considers LRAs deployed in homogeneous containers. However, our approach can be easily extended to incorporate heterogeneous clusters or container resource demands by embedding machine/container properties into the RL state, e.g., machine configurations and container resource requests.

Implementations on Other Container Orchestrations George is currently implemented on Docker Swarm [12], for its simplicity. George's scheduling workflow, i.e., in-place model training, the ILP-based filter, PPPO training and transfer learning techniques can also be extended to other frameworks like Kubernetes [17].

8 Conclusion

In this paper, we first examined the inefficiency of existing LRA scheduling in coping with the operation constraints. We then presented George, an intelligent end-to-end general-purpose scheduler driven by a novel projection-based proximal policy optimization (PPPO) algorithm tailored to LRA scheduling. To enforce hard constraints, George filters the action space by excluding those that violate the hard constraints through verification of the feasibility with an ILP formulation, followed by RL-base decision making. To satisfy the soft constraints, the proposed PPPO algorithm projects the RL model update to a "safe zone", such that the violation of soft constraints can be guaranteed within a pre-defined threshold. George applies transfer learning techniques by taking advantage of the similarity between LRA scheduling events, which significantly reduces the model training time. Our EC2 deployment demonstrated that George can guarantee the operation constraints without compromising the container scheduling performance, and can scale to a large cluster running thousands of containers.

Acknowledgement

We would like to thank our shepherd, Dilma Da Silva, and the anonymous reviewers for their valuable feedback that helps improve the quality of this work. This research was supported in part by RGC GRF grants under the contracts 16213120, 16207818 and 16209120 and RGC RIF grant under the contract R6021-20.

References

[1] 2018. Getting Started with A/B Testing. https://developer.amazon.com/blogs/appstore/post/Tx27HL6EMW36UCL/getting-started-with-a-b-testing.

[2] 2019. Aurora. http://aurora.apache.org.

[3] 2019. Marathon: A container orchestration platform for Mesos and DC/OS. http://mesosphere.github.io/marathon/.

[4] 2021. Alibaba production cluster data. https://github.com/alibaba/clusterdata.

[5] 2021. Amazon Elastic Compute Cloud (Amazon EC2). https://aws.amazon.com/ec2.

[6] 2021. AMAZON WEB SERVIES, INC. AWS Lambda: Serverless computing. https://aws.amazon.com/cn/lambda/.

[7] 2021. Apache flink. https://flink.apache.org/.

[8] 2021. Apache HBase. https://hbase.apache.org/.

[9] 2021. Apache Kafka. https://kafka.apache.org/.

[10] 2021. Apache MXNet. http://mxnet.incubator.apache.org/.

[11] 2021. Apache Storm. https://storm.apache.org/.

[12] 2021. Docker swarm. https://github.com/docker/swarm.

[13] 2021. George GitHub Repository. https://github.com/lwangbm/george-LRA-scheduler.

[14] 2021. George: Learning to Place Long-Lived Containers in Large Clusters with Operation Constraints (Appendix). https://1drv.ms/b/s!Ar8AJEjgeqwBbSgFMiE4LRhTs9o?e=tKlCBb.

[15] 2021. Google production cluster data. https://github.com/google/cluster-data.

[16] 2021. hashlib: Secure hashes and message digests. https://docs.python.org/3/library/hashlib.html.

[17] 2021. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/.

[18] 2021. Locust: an open source load testing tool. https://locust.io/.

[19] 2021. Memcached. https://memcached.org/.

[20] 2021. Model Server for Apache MXNet. https://github.com/awslabs/mxnet-model-server.

[21] 2021. PySceneDetect: Python and OpenCV-based scene cut/transition detection program & library. https://github.com/Breakthrough/PySceneDetect/.

[22] 2021. Redis: an open source, in-memory data structure store. https://redis.io/.

[23] 2021. Redis-benchmark. https://redis.io/topics/benchmarks.

[24] 2021. Solr: An open-source enterprise search platform built on Apache Lucene. https://solr.apache.org.

[25] 2021. wrk: Modern HTTP benchmarking tool. https://github.com/wg/wrk.

[26] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *Proc. USENIX OSDI*.

[27] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. 2017. Constrained policy optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. 22–31.

[28] Yixin Bao, Yanghua Peng, and Chuan Wu. 2019. Deep Learning-based Job Placement in Distributed Machine Learning Clusters. In *Proc. IEEE INFOCOM*.

[29] Andrew G Barto and Sridhar Mahadevan. 2003. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems* 13, 1-2 (2003), 41–77.

[30] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[31] Luiz A Celiberto Jr, Jackson P Matsuura, Ramón López De Màntaras, and Reinaldo AC Bianchi. 2010. Using transfer learning to speed-up reinforcement learning: a cased-based approach. In *2010 Latin American Robotics Symposium and Intelligent Robotics Meeting*. IEEE, 55–60.

[32] Yanpei Chen, Sara Alspaugh, and Randy Katz. 2012. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1802–1813.

[33] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. 2011. The case for evaluating mapreduce performance using workload suites. In *2011 IEEE 19th annual international symposium on modelling, analysis, and simulation of computer and telecommunication systems*. IEEE, 390–399.

[34] Yue Cheng, Zheng Chai, and Ali Anwar. 2018. Characterizing Co-Located Datacenter Workloads: An Alibaba Case Study. In *Proceedings of the 9th Asia-Pacific Workshop on Systems* (Jeju Island, Republic of Korea) *(APSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 12, 3 pages. https://doi.org/10.1145/3265723.3265742

[35] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proc. ACM SoCC*.

[36] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System.. In *Proc. USENIX NSDI*.

[37] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[38] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 77–88.

[39] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *ACM SIGARCH Computer Architecture News*, Vol. 42. ACM, 127–144.

[40] Thomas G Dietterich. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of artificial intelligence research* 13 (2000), 227–303.

[41] Francesco Cardinale et al. 2018. ISR. https://github.com/idealo/image-super-resolution.

[42] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: scheduling of long running applications in shared production clusters. In *Proc. ACM EuroSys*.

[43] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. 2017. Swayam: distributed autoscaling to meet SLOs of machine learning inference services with resource efficiency. In *Proc. ACM/IFIP/USENIX Middleware*.

[44] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who limits the resource efficiency of my datacenter: an analysis of Alibaba datacenter traces. In *Proc. ACM IWQoS*.

[45] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 34–50. https://doi.org/10.1145/3132747.3132749

[46] John K Karlof. 2005. *Integer programming: theory and practice*. CRC Press.

[47] Eugene L Lawler and David E Wood. 1966. Branch-and-bound methods: A survey. *Operations research* 14, 4 (1966), 699–719.

[48] Qixiao Liu and Zhibin Yu. 2018. The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace. In *Proc. ACM SoCC*.

[49] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 450–462.

[50] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the cloud: An analysis on alibaba cluster trace. In *Proc. IEEE Big Data*.

[51] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[52] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. 2018. Variance reduction for reinforcement learning in input-driven environments. *arXiv preprint arXiv:1807.02264* (2018).

[53] Daniel A Menascé. 2002. TPC-W: A benchmark for e-commerce. *IEEE Internet Computing* 6, 3 (2002), 83–87.

[54] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[55] 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*. 561–577.

[56] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proc. ACM Eurosys*.

[57] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proc. USENIX ATC*.

[58] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[59] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. 2015. Apache tez: A unifying framework for modeling and building data processing applications. In *Proc. ACM SIGMOD*.

[60] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust Region Policy Optimization. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 37)*, Francis Bach and David Blei (Eds.). PMLR, Lille, France, 1889–1897.

[61] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[62] Bikash Sharma, Victor Chudnovsky, Joseph L Hellerstein, Rasekh Rifaat, and Chita R Das. 2011. Modeling and synthesizing task placement constraints in Google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. 1–14.

[63] Ruben Solozabal, Josu Ceberio, and Martin Takáč. 2020. Constrained combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:2006.11984* (2020).

[64] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction.* MIT press.

[65] Matthew E Taylor, Gregory Kuhlmann, and Peter Stone. 2008. Autonomous transfer for reinforcement learning.. In *AAMAS (1)*. Citeseer, 283–290.

[66] Matthew E Taylor and Peter Stone. 2009. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research* 10, 7 (2009).

[67] Chen Tessler, Daniel J. Mankowitz, and Shie Mannor. 2019. Reward Constrained Policy Optimization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

[68] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. 2017. Phoenix: A constraint-aware scheduler for heterogeneous datacenters. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 977–987.

[69] Huangshi Tian, Minchen Yu, and Wei Wang. 2018. Continuum: A Platform for Cost-Aware, Low-Latency Continual Learning. In *Proc. ACM SoCC*.

[70] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proc. ACM EuroSys*.

[71] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proc. ACM SoCC*.

[72] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proc. ACM Eurosys*.

[73] Luping Wang, Qizhen Weng, Wei Wang, Chen Chen, and Bo Li. 2020. Metis: Learning to Schedule Long-Running Applications in Shared Container Clusters at Scale. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM/IEEE.

[74] H. Wu, W. Zhang, Y. Xu, H. Xiang, T. Huang, H. Ding, and Z. Zhang. 2019. Aladdin: Optimized Maximum Flow Management for Shared Production Clusters. In *Proc. IEEE IPDPS*. 696–707.

[75] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: SQL and rich analytics at scale. In *Proc. ACM SIGMOD*.

[76] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 607–618.

[77] Tsung-Yen Yang, Justinian Rosca, Karthik Narasimhan, and Peter J Ramadge. 2020. Projection-Based Constrained Policy Optimization.. In *ICLR*.

[78] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. ACM SOSP*.

[79] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI 2: CPU performance isolation for shared compute clusters. In *Proc. ACM Eurosys*.