This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446

CARL: Cost-optimized Online Container Placement on VMs using Adversarial Reinforcement Learning

Prathamesh Saraf Vinayak, Saswat Subhajyoti Mallick, Lakshmi Jagarlamudi *Senior Member, IEEE*, Anirban Chakraborty and Yogesh Simmhan *Senior Member, IEEE*

***Abstract*—Containerization has become popular for the deployment of applications on public clouds. Large enterprises may host 100s of applications on 1000s containers that are placed onto Virtual Machines (VMs). Such placement decisions happen continuously as applications are updated by DevOps pipelines that deploy the containers. Managing the placement of container resource requests onto the available capacities of VMs needs to be cost-efficient. This is well-studied, and usually modelled as a multi-dimensional Vector Bin-packing Problem (VBP). Many heuristics, and recently machine learning approaches, have been developed to solve this NP-hard problem for real-time decisions. We propose CARL, a novel approach to solve VBP through Adversarial Reinforcement Learning (RL) for cost minimization. It mimics the placement behavior of an offline semi-optimal VBP solver (teacher), while automatically learning a reward function for reducing the VM costs which out-performs the teacher. It requires limited historical container workload traces to train, and is resilient to changes in the workload distribution during inferencing. We extensively evaluate CARL on workloads derived from realistic traces from Google and Alibaba for the placement of 5k–10k container requests onto 2k–8k VMs, and compare it with classic heuristics and state-of-the-art RL methods. (1) CARL is *fast*, e.g., making placement decisions at ≈ 1900 requests/sec onto $8,900$ candidate VMs. (2) It is *efficient*, achieving $\approx 16\%$ lower VM costs than classic and contemporary RL methods. (3) It is *robust* to changes in the workload, offering competitive results even when the resource needs or inter-arrival time of the container requests skew from the training workload.**

***Index Terms*—Container placement, Cloud scheduling, Resource Optimization, Adversarial Reinforcement Learning, ML for Systems**

I. Introduction

ENTERPRISES are increasingly using containers to package and deploy applications developed using microservices and serverless paradigms onto clouds. Adoption of DevOps principles means that any changes to an application can trigger the (re)deployment, placement and launch of 100s of containers. DevOps admins in large enterprises may handle 1000s of such daily deployment requests, with 10s of containers spawned for each application [1]. Further, Function as a Service (FaaS) providers can launch 200k container requests within minutes onto VMs to handle incoming load [2], while enterprises ranging from gaming to finance have adopted Kubernetes to orchestrate 1000s of containerized applications scaling up and down to millions–billions of requests per day [3]. These containers need to be efficiently provisioned on a pool of diverse Virtual Machines (VMs) of that enterprise, either as reserved instances or on-demand ones on public or private clouds.

All authors are with the Indian Institute of Science, Bangalore, India. Email: {prathameshs, saswatm, jlakshmi, anirban, simmhan}@iisc.ac.in

Directly managing the placement logic of containers for an enterprise, rather than using public cloud container services like AWS Elastic Kubernetes Service (EKS) and Google Kubernetes Engine (GKE), can be more cost-efficient if done intelligently.

1) Problem: Placement decisions must maintain the container's performance and reduce VM usage costs, while taking the decision in near real-time. Specifically, given the resource needs for container requests arriving for placement and the pool of active or available VMs along with their resource capacities, we need to place the containers onto the fewest possible VMs so as to minimize any increase in the VM costs. This placement problem is usually formalized as a multi-dimensional *Vector Bin Packing (VBP) problem* [4]. Such problems pack a diverse set of *requests* – containers described by a vector of their resource needs (e.g., CPU, memory, disk, etc.), onto *heterogeneous bins* (VMs) – specified by vectors of resource capacities and bin prices to minimize the cumulative bin costs while not exceeding any bin's resource capacity.

2) Challenges: Solving the bin packing problem *optimally* is NP-Hard [5], and not practical for placing 100s of requests arriving each second in near real-time. Optimal solutions, typically solved using Integer Linear Programming (ILP) [5], [6], can take hours even for $10s$ of requests, as we show later.

Further, while the computing time for "optimal" solutions may be modest for low arrival rates (few requests per minute), these are only optimal at a point in time (e.g., VP_i, discussed later), and may be globally sub-optimal across time as new requests arrive. Truly optimal solutions (e.g., VP_o) require perfect knowledge of all future requests, which is not feasible in real-world settings. This makes "oracles" impractical for online decision-making.

Numerous (sub-optimal) *heuristics* have been developed for fast placement [7], [8], ranging from classic First Fit Decreasing (FFD), Best Fit (BF) and variants like Tetris [4], [9] to more recent machine learning based approaches [10], [11], including from cloud providers like Microsoft Azure [12] and Google Borg [13]. However, they are designed for alternative metrics than cost (e.g., improving makespan, energy or utilization), are not resilient to changes in the incoming trace, and/or offer a worse solution than our proposed work.

Recently, *Reinforcement Learning (RL)* has gained popularity [14] for making scheduling decisions in clusters and cloud

0000–0000/00$00.00 © 2025 IEEE

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446

data centers [10], [11]. While training the RL model takes time, it is done offline and the inferencing time for placement decisions is fast – suitable for real-time use. Importantly, RL uses domain-specific reward functions to accelerate learning, and this requires less training data compared to deep learning methods [15]. But designing a good reward function to capture the optimization objectives and constraints, and tuning the learning model and hyper-parameters, can be non-trivial.

In summary, we need a *fast, efficient and robust method* to make real-time placement decisions for 100s of container requests arriving per seconds onto heteorogeneous VMs.

3) Approach: In this paper, we solve the placement problem using an adversarial RL approach which automatically learns the reward function, guided by an offline near-optimal "teacher" (or expert). We first map the problem to the popular Advantage Actor-Critic (A2C) [16] RL strategy, and enhance it with a custom reward function we develop based on the VM costs and introduce novel action-clustering to address the high-dimensionality action space. Next, we improve this with Generative Adversarial Imitation Learning (GAIL) [17], where the actor emulates the placement decisions of a semi-optimal VBP solver (teacher) [5], to automatically learn a dynamic reward function. This combined strategy we propose is called CARL, Cost-optimized placement using Adversarial Reinforcement Learning. This uses fewer training samples and limited human expertise to design a reward function, and even out-performs the teacher. *To our knowledge, this is the first RL method guided by an ILP solver teacher to learn a placement policy through generative adversarial imitation learning.*

4) Discussion: This article offers a novel solution to the known problem of online scheduling, in this case motivated by the need for realtime container placement onto elastic VMs. Such NP-hard online scheduling problems have been well-studied for decades through approximate heuristics [9], [18]. Even RL techniques have recently been applied with promising results [10], [19]. Given the absence of a provably better solution, any new method draws skepticism on whether it is truly better. Any candidate solution requires demonstration using extensive empirical experiments on realistic workloads, and detailed comparison with state-of-the-art (SOTA) techniques. We attempt to do so in this paper to convincingly demonstrate: (1) The *fast* decisions of CARL for online placement, (2) Its better *cost efficiency* relative to two classic and two SOTA RL methods, and (3) Its *robustness* to change the placement request distribution relative to training.

Given that empirical validation of 100k of container requests is cost and resource prohibitive (e.g., our simulations use 1000s of VMs), we limit ourselves to validation using realistic simulations of real-world container request traces. Also, containers and micro-VMs can be spun up and down in seconds and used extensively for resource sandboxing, allowing our results to be translated into practise. Co-scheduling of dependent containers in a complex serverless deployment is complementary to this problem and left to future work.

Further, while the direct application of this technique is for the important challenge of online container placement in cloud data centers, the problem formulation and solution approach itself is general enough to be applied to other real-time resource optimization problems.

5) Contributions: We formulate the container placement problem as the well-known 4D vector bin-packing problem to achieve VM cost minimization (§ III), and make these key novel contributions:

1) We design an initial solution using Actor-Critic RL with a custom reward function, and incorporate a unique action clustering approach (A2C2) to improve its performance for high-dimensionality action spaces (§ IV-A).
2) We propose CARL, a novel adversarial RL method to learn a placement strategy using adversarial learning of a dynamic reward function from a semi-optimal ILP teacher, over a modest number ($1k$–$5k$) of training samples (§ IV).
3) We conduct detailed comparative evaluations of CARL against two classic (VPSolver [20], Tetris [4]) and two RL-based SOTA approaches (A2C2 [16] and Behavior Cloning (BC) [21]) for real-time placement decisions on several realistic and heterogeneous workloads with $1k$–$10k$ requests sampled from Google and Alibaba traces for placement onto $2k$–$8k$ diverse VMs (§ V).

CARL reduces costs on average by 16.3% when compared with classic heuristics and by 15.1% over the A2C2 and BC RL methods, while making placements decisions at ≈ 2000 requests/sec. It is also resilient to changes in the input workload distribution if it deviates from the trained traces in terms of arrival time and resource needs.

The rest of the article is organized as follows: § II compares our work against related works, § III offers the problem formulation, § IV proposes the CARL solution, § V describes the experiment workloads and analyzes the results, and § VI provides the conclusions.

II. Background and Related Work

There is extensive research into placement problems for cloud resources, including VMs onto servers and applications, containers, and micro-services onto VMs. Several detailed surveys explore them [8], [10], [18], [19], [22]. This section discusses and contrasts these efforts along three approaches taken: optimal, heuristic and learning based methods.

A. Optimal Methods

Vector Bin Packing (VBP) is the natural formulation of the multi-dimensional placement problem, with numerous techniques to optimally solve this NP-Hard problem [5], [6], [9]. Bartók et al. [23] use a branch and bound strategy along with VM-specific heuristics to solve for VM placement onto bare-metal servers. Anand et al. [24] use ILP for VM placement in accordance with SLAs. The drawback of these is the long computing time, linearly increasing with the dimensionality and the number of VMs, even with parallelization, and making them unsuitable for real-time scheduling.

A recent promising work, *VPSolver* [5], proposes an arc-flow formulation with graph compression. It reports a nominal time complexity between $\Theta(n^2)$—$\Theta(n^{2.5})$ and solves previously unsolved benchmarks [25].

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446

The algorithm has several steps. It generates an arc-flow graph for the vector packing problem which represents all possible packings of items into bins. This graph is then compressed by identifying and merging equivalent states to reduce its size while preserving the optimal solutions. The compressed graph is used to generate a Mixed-Integer Programming (MIP) model, which itself is solved using a general-purpose solver like Gurobi [26]. This returns the optimal packing of items.

While being competitive, the performance of VPSolver is tied substantially to the size of the graph replicating all valid patterns. As items that can fit in a resource bin increases, the execution time increases due to the creation of long packing patterns. As we show, it takes hours to place 100s of containers onto 1000s of VMs. Also, VPSolver gives the optimal solution (which we call VP_o) *only if it has access to the entire (future) request trace* of containers and the set of all VMs. But this is infeasible when requests stream in without a pattern, and placements have to be decided online. To overcome this, we propose an incremental variant (VP_i, see § IV-C), which is a *semi-optimal baseline* for comparison in our experiments and also serves as a teacher for CARL.

B. Heuristic Methods

Heuristics offer fast decisions, but their approach tends to be *ad hoc* and the quality of placement well suited for specific workloads. First Fit Decreasing (FFD) and Best Fit (BF) are classic greedy heuristics that provide good solutions, but to a 1D problem [7]. Others like Min-Min and Max-Min are sub-optimal and also do not reflect the multi-dimensional nature of the container and VM resources [27], [28].

Tetris [4] is an efficient multi-dimensional bin packing algorithm that avoids resource fragmentation and over allocation. Tetris computes an alignment score between the task and the machine, which is a dot product between the vector of machine's available resources and the task's requirement. From the set of tasks to be considered for placement, the task that has the highest alignment score is placed on the machine. But it does not consider the deployment cost as a first-class entity, which CARL does. We use Tetris as a *heuristic baseline* for comaprison in our experiments.

Dominant Resource Fairness (DRF) [29] has been developed as a policy to allocate different resource types to containers in a system, and has been extended [30] to accommodate multiple servers and minimize energy usage. We differ from these by considering multi-dimensional resource vectors from containers being matched to VM bins and aim for a cost-minimization across dynamic online placement requests. Others [31] propose VM placement onto physical hosts in clouds, tuned for HPC applications, and use a single *m1.small* VM for evaluation. We formulate and solve a more general problem with diverse bin categories and request demands.

Cloud providers also use heuristics for their data center scheduling decisions. Microsoft Azure uses a rule-based scheduler for VM placement onto servers, with customization for workload patterns [12]. However, it is designed for homogeneous servers, while we operate over heterogeneous VMs common in enterprises. Google's Borg cluster manager [13] schedules jobs onto compute resources to maximize the utilization, by combining admission control, efficient task-packing, over-commitment and machine sharing with process-level performance isolation. CARL also indirectly enhances the packing efficiency. But it takes a user, rather than cloud provider, view and aims to minimize costs to rent the VMs without over-provisioning resources and affecting performance. Our evaluation uses $\approx$ 9K VMs, similar to Borg, and in fact uses a workload trace from Google that pre-dates Borg traces.

C. Learning-based Methods

Reinforcement Learning (RL) has been more successful [10], [32] for scheduling problems since they explore the search space better, and learn from their mistakes using a reward function compared to *Deep Neural Networks* [33]. *Q-learning* is a form of RL used for dynamic job scheduling [14] and has been combined with an Actor-Critic method for micro-service resource allocation over scientific workflows [34], and HPC batch scheduling to maximize resource utilization [35]. Here, the *actor*, or Policy network, selects actions to take based on the current state, such as choosing a VM for container placement while the *critic*, or Value network, evaluates the state's long-term value. This has been extended to Advantage Actor-Critic (A2C) [16], where an additional *advantage* function refines the actor's policy by guiding the relative desirability of different actions in different states. This can lead to actions that offer higher-than-expected rewards.

However, these approaches can still result in poor training and performance when subjected to large dimensional state-action spaces, particularly relevant in our problem where we have 1000s of VMs for placing in. We addresses this limitation using *action clustering* and enhance the strategy using our proposed *Advantage Actor-Critic with Clustering (A2C2)* method, where groups of VMs with similar available capacity are clustered.

Further, policy optimization methods such as PPO have shown to converge faster than Q-learning [36]. So, A2C2 uses PPO for policy optimization by the advantage function, resulting in improved training stability, sample efficiency, and faster convergence. This enhanced A2C2 strategy serves as a *state-of-the-art RL baseline* for us.

Imitation learning is another form of RL where a policy is learned from expert (teacher) demonstrations rather than a reward signal or objective function. Wang et al. [37] use imitation learning to predict job remaining time in HPC clusters, and uses it to preempt jobs as an expert for better training. Similarly, *Behavior Cloning (BC)* [21], a form of imitation learning, uses supervised learning on a set of expert demonstrations that are collected offline to learn the expert's behavior. Guo et al. [38] use BC, with Shortest Job First (SJF) as their expert, and a Convolutional Neural Network (CNN) for cloud scheduling. However, BC suffers from covariate shift [39], which makes it difficult to generalize and recover from errors. Since, BC is only trained on expert data and does not interact with the environment, it does not know how to handle situations that deviate from expert. We use BC with VPSolver as the expert as yet another *state-of-the-art RL baseline* for comparison.

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446

Recently, *Generative Adversarial Networks (GANs)* [40] have been used to learn the RL policy directly from the training data. GAIL [17] is such an adversarial RL technique to fit distributions of states and actions that define an expert's behavior by interacting with the environment. This allows GAIL to learn robust control policies that can handle a wider range of situations. Pan et al. [41] use permutation as an adversary to generate worst case problem instances for solving a 3D bin packing problem to improve policy robustness for worst-case scenarios while maintaining acceptable performances for nominal cases. CARL uses an adversarial method based on GAIL to train our A2C2 strategy and uses VPSolver as the expert. During training, CARL generates multiple data distributions by interacting with the environment as it tries to match the expert's distribution. This makes it resilient to skews in the input workload.

III. Problem Formulation

We define the placement problem of containers (or tasks) onto VMs (or bins) as a *multiple-choice VBP problem* in 4 dimensions: CPU cores, memory, disk capacity and network bandwidth [1].

A *bin* (VM) with ID j is represented as a vector with these available resource capacities, $\langle c^j_{cpu}, c^j_{mem}, c^j_{disk}, c^j_{nw}\rangle$, and a price ρ^j associated for its usage for a unit time period (e.g., billing per second or hour). In public clouds, this cost is typically a function of the resource *capacities*. An incoming *item* (container or task) with ID i is to be assigned a bin and is also represented as a vector with 4 dimensions indicating the *weights* (resource requirements), $\langle w^i_{cpu}, w^i_{mem}, w^i_{disk}, w^i_{nw}\rangle$ for this container. The binary variable $x_{i,j}$ indicates if the i^{th} item is placed into the j^{th} bin, i.e,

$x_{i,j} = 1$ If Item i is assigned to bin j
$x_{i,j} = 0$ Otherwise

Items continuously arrive over time for placement, with their current arrival rate depending on the workload. When a *batch* k with n items accumulated over a defined period of time (e.g., 1 second) arrives, the items need to be assigned among the m available bins such that the *total cost of the active bins* is minimized. A bin is *active* if it has at least one item assigned to it. The cost of the active bins after assigning a batch k is:

$$C_k = \sum_{j=1}^{m} \rho_j \mid \exists x_{i,j} = 1 \ \forall \ i \text{ after assigning items in batch } k$$

This is based on the billing policy on public clouds, where the acquisition of a VM causes it to be billed, irrespective of the load on the VM.

Minimizing C_k is the **optimization goal**, subject to the following **constraints**:

- Each item can be placed in only one bin, $\sum_{j=1}^{m} x_{i,j} = 1 \ \forall i$
- Items in a bin cannot exceed the bin's available capacity, $\sum_{i=1}^{n} w^i_d x_{i,j} \leq c^j_d \ \ \forall j \in 1 \ldots m, d \in \{cpu, mem, disk, nw\}$.

This placement decision also has to be done fast, to keep up with the input rate of items and to avoid startup delays.

[1]In the rest of the paper, bins and VMs are used interchangeably, as are containers and tasks.

When items in a batch are assigned to bins, the capacity available for the bins changes. Similarly, a container may be released by the user after it has served its purpose, and its bin's capacity will correspondingly increase for the next batch.

Discussion: VM cost is accrued only for active bins when at least one item is assigned to it. We assume an arbitrarily large number of bins of different VM types available for provisioning, but only the active bins are billed. In practice, a newly active bin will cause a VM to be acquired on-demand, while a bin with no items assigned causes that VM to be turned off. While public clouds may impose minimum retention periods for VMs, these are small enough (e.g., 60 seconds for AWS EC2) to not affect costs. There can be VM boot-up times that delay the placement. But keeping one or a few warm empty VMs can avoid this cold-start, with only incremental costs when 100s of VMs are involved.

We consider the peak resource usage for a container here. But this can be made more sophisticated by provisioning for the 90^{th} or 95^{th} percentile. Additionally, although, we have a strict bound on the VM capacity, practical systems may over-commit VM resources for further cost-efficiency by, e.g., 10%. These strategies can be achieved by applying scaling factors on the container resources (e.g., $0.95\times$) or VM capacities ($1.1\times$). While we consider four common resource types here, our proposed framework is generalizable to accommodate any number of additional resource types, e.g., GPUs. Once a container is placed in a VM, it is not migrated out during its lifetime and only removed once its usage is completed. We also do not consider container migrations, which are more complex that can cause service disruption and not routine in an enterprise setting which values simplicity and scaling. These assumptions are close enough to reality and cover a large space of enterprise deployments to make the proposed solution cloud and platform neutral, and widely applicable.

IV. Adversarial Imitation Learning with CARL

Our proposed online cost-optimized *Container placement on VMs using Adversarial Reinforcement Learning (CARL)* strategy builds upon the Advantage Actor Critic with Clustering (A2C2) RL method discussed earlier and complemented by Generative Adversarial Imitation Learning (GAIL) to learn a policy. Here, we describe the RL environment used by A2C2 and CARL, followed by a detailed description of CARL.

A. RL Environment

Our RL environment is defined by these key components.

a) State Space (S): The state space represents the observations of the placement environment. It is defined as a matrix $S_{(m+1)\times 5}$, where m is the number of available VMs. Each VM $j \in [1..m]$ is described by a tuple $\langle x^j, c^j_{cpu}, c^j_{mem}, c^j_{disk}, c^j_{nw}\rangle$, where x^j is a binary status indicator denoting if the VM is active ($x = 1$) or turned-off ($x = 0$) while the rest c^j_* indicate the available capacity of its CPU cores, memory, disk storage and network bandwidth. The $(m+1)^{th}$ row represents the resource requirements of the task, and its status indicator is always set to zero. Min-max normalization is applied across all rows for all dimensions except for the status.

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446

b) Action Space (A): The discrete action space of size $(m \times 1)$ gives the set of m possible actions, where each action corresponds to selecting a VM j out of a total of m available VMs to place the incoming container request onto.

c) Reward (R): The reward is the feedback the environment gives the RL agent for taking a particular action. CARL, being adversarially trained, does not need an explicit reward function to be defined by the user (as described later). However, A2C2 uses the reward function described below, which aligns with the objective of VM cost minimization.

When the j^{th} VM has been selected for placement of a task and is currently active, its status will be set to $x^j = 1$; if the VM was previously inactive (unused), its status would change from 0 to 1. Let ρ^j be the cost to keep the VM j active per unit-time. In practise, this is the actual cost of that VM type in the public cloud [42].

The reward function for A2C2 is the negative sum of the total cost of all *active* machines divided by the total cost of all the m *available* VMs in the environment. Formally:

$$Reward \quad = \quad -\frac{\sum_{j=1}^{m} x^j . \rho^j}{\sum_{j=1}^{m} \rho^j}; \qquad (1)$$

This is the total reward an agent can get when placing a task onto a machine. It includes the cost of already active machines, which is necessary for the placement decision to capture the earlier tasks and their completions too. Dividing the current cost of active machines by the total cost helps in normalizing the reward, which makes reward signals consistent and comparable across different episodes.

d) Episode termination: An episode is a sequence of interactions that the agent has with the environment. It begins with an initial state and continues until certain conditions are met. An episode terminates when all the placement demands are met or an invalid action is selected, e.g., the agent attempts to over-commit a VM; in such cases, a heavy negative reward C is applied, where we set $C = -1000$ in our experiments.

e) Policy π: The policy is a mapping learned by the RL agent to maximize its reward across different episodes within the environment. We use A2C [16] with PPO [36] as the learning policy for both A2C2 and CARL. Further, we extend the A2C+PPO policy with action clustering (A2C2), where VMs are grouped based on their current available capacity. This reduces the action space from 1000s of active or inactive VMs to the number of distinct VM capacities. Only one VM (preferably active) is considered from each cluster for placement. E.g., if we have $10K$ VMs with only four unique VM capacity vectors, action clustering will reduce the action space from $10K$ to 4. We report that our A2C2 enhancement offers a 26% reduction in the average placement cost for the Google workload introduced later, compared to the vanilla A2C+PPO available in literature.

The clustering approach in A2C2 also accounts for capacity changes to the VM due to the arrival and departure of the tasks that are placed on them. As tasks running on a VM reach their end time and complete their execution, the capacity of the VM increases. If all tasks on a VM finish, the VM stops being billed and its capacity becomes fully available, as in a new VM. In our simulation evaluation described later, we cluster VMs done just in time, when a scheduling request arrives. At that instance, VMs with a similar available capacity, including VMs that do not have any task assigned, are grouped together. When a VM is to be selected from a cluster for placement, active VMs with existing tasks are given preference.

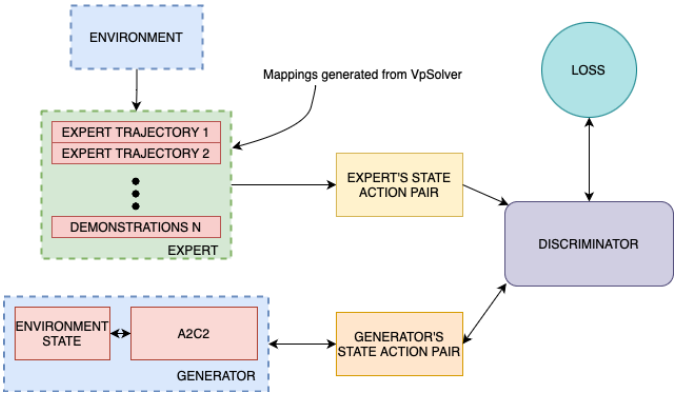


Figure 1: Architecture of CARL

B. CARL

Our proposed CARL strategy uses GAIL [17] as the adversarial training technique to define the learning policy for our RL solution. Here, the RL agent utilizies concepts analogous to GANs [40] to observe the behavior of an expert (teacher) and infers the underlying decision-making policy. The expert for CARL, as we discuss next, is the semi-optimal incremental vector bin packing solver, VPSolver [20]. This approach serves as an alternative to the user explicitly defining a reward function, as was done for A2C2.

CARL harnesses generative adversarial training to fit distributions of the states and the actions that define the behavior of the expert. As shown in Figure 1, the expert's trajectories (state–action pairs) are generated by iterating the solution provided by VPSolver for the current placement set (mapping of task to VM) through the RL environment. Each trajectory consists of the current environment state (VM availability) and the action taken (VM selected to place a given task). Once the set of expert trajectories is generated, the goal is to train the A2C2 to generate state–actions that are similar to the expert trajectories. This is achieved by using an adversarial learning strategy that involves training a suitable discriminator, similar to ones used in GANs. The discriminator network in CARL helps distinguish between trajectories created by the generator (policy) network and ones obtained from the expert.

1) Algorithm Description: Algorithm 1 describes the overall working of CARL. At the outset, the parameters for the Actor and Critic networks and an additional Discriminator network are initialized (line 1). The Actor network is responsible for the policy, i.e., action selection, while the Critic network estimates the value of states. The discriminator is a binary classifier that distinguishes between the expert's actions and the agent's actions. The expert's trajectories as obtained from VPSolver are loaded (line 2) to provide a reference point of learning for agent's behavior. Next, we iterate over multiple episodes until E_{max} is reached (line 3). We initialize the set of empty VMs that forms our initial state distribution s_0 and a trajectory buffer to store the generated transitions (line 4-5).

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446

Algorithm 1 Cost optimized Adversarial Reinforcement Learning (CARL)

1: Initialise parameters θ for Actor and ϕ for Critic, and ψ for Discriminator
2: Load expert trajectories $\tau' \leftarrow \{(s', a')\}$
3: **for** episode $E \leftarrow 1..E_{max}$ **do**
4: Initialize trajectory buffer, $\tau \leftarrow \{\}$
5: Initialize the initial state distribution of VMs, s_0
6: **for** timestep $t \leftarrow 0..T_{max} - 1$ **do**
7: Sample action from distribution, $a_t \leftarrow \pi_\theta(a_t|s_t)$
8: Execute a_t. Observe next state s_{t+1} and environment reward r_{t_env} using policy θ.
9: Store transition, $\tau \leftarrow (s_t, a_t, r_{t\_env}, s_{t+1})$
10: Compute discriminator reward:
$$r_D \leftarrow -\log(1 - D_\psi(s_t, a_t) + \epsilon)$$
11: Update total reward, $r_t \leftarrow r_{t\_env} + r_D$
12: Compute advantage estimate:
$$A_t \leftarrow r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$$
13: Compute the ratio between the current policy (π_θ) and previous policy ($\pi_{\theta_{old}}$):
$$r(\theta) \leftarrow \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$
14: Compute PPO objective:
$$J(\theta) \leftarrow \min(r(\theta)A_t, \text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon)A_t)$$
15: Compute Actor network's (policy) gradients: $\nabla_\theta J(\theta)$
16: Update θ_{old} with θ, $\theta_{old} \leftarrow \theta$
17: Update Actor parameters, $\theta \leftarrow \theta - \alpha_\theta \nabla_\theta J(\theta)$
18: Compute Critic loss, $J(\phi) \leftarrow (V_\phi(s_{t+1}) - V_\phi(s_t))^2$
19: Update Critic, $\phi \leftarrow \phi - \alpha_\phi \nabla_\phi J(\phi)$
20: **end for**
21: Sample batch of agent trajectories, $\tau_A \subset \tau$
22: Sample batch of expert trajectories, $\tau'_E \subset \tau'$
23: Compute discriminator loss: $L(\psi) \leftarrow \mathbb{E}_{(s,a)\sim\tau_A}[\log(D_\psi(s, a))] + \mathbb{E}_{(s_E,a_E)\sim\tau'_E}[\log(1 - D_\psi(s_E, a_E))]$
24: Update the discriminator parameters: $\psi \leftarrow \psi - \alpha_\psi \nabla_\psi L(\psi)$
25: **end for**

For each for episode, take T_{max} number of steps (line 6). At each time step, several operations are performed.

An action is selected using the policy defined by the Actor network, the action is performed, and the environment returns a reward and the new state (lines 7–9). These transitions are stored in the trajectory buffer.

We then calculate the discriminator's reward as a function of the state and action to encourage the agent to imitate the expert's behavior (line 10). The discriminator reward is combined with the environment reward to form the total reward (line 11). This combined reward is used in subsequent actor and critic updates.

We then compute the advantage estimate (line 12), which measures how much better the action is compared to the average action at that state. Next, we compute the ratio of the current and previous policy probabilities (line 13), which is used in the surrogate PPO objective function (line 14) to ensure that the new policy does not deviate too much from the old policy and mitigates the risk of large changes. The gradient is computed with respect to the policy parameters θ (line 15) and the actor network is updated (line 17).

We compute the Critic loss (line 18) and update the Critic network's parameters (line 19). The loss is the squared difference between the estimated value of the current state and the actual return, while the gradient of the Critic network's loss with respect to the value function parameters ϕ is used to update the parameters ϕ.

After processing all transitions, we sample batches of agent and expert trajectories (lines 21–22). We then compute the discriminator loss (line 23), which is designed to compare between the expert's action with that of the agent's action. Given (s, a) as a tuple of state–action pair, generated by our agent, the discriminator is trained to output a high probability for the expert's trajectories (s_E, a_E), and a low probability for trajectories generated by the agent (s, a). The objective of this adversarial training is to reach a point where the discriminator cannot distinguish between the RL agent's actions and the expert-generated assignment. We then update the parameters of the discriminator network using gradient descent on the discriminator's loss (line 24). At this point, the policy network has effectively learned to follow the expert's behavior, i.e, the generated actions are similar to that of the expert.

2) Key Functions: The *Advantage function* A_t measures how much better an action is compared to the average action at that state. It is used to compute the policy gradient as the difference between the actual return and the estimated value of the state. The actual return is the immediate reward plus the discounted estimated value of the next state, and the estimated value of the state is given by the Critic network. The Advantage function is used to scale the policy gradient, encouraging actions with positive advantage (better than average) and discouraging actions with negative advantage (worse than average). This helps the policy to improve over time. The Advantage function is a key component of both A2C and PPO algorithms. It helps to reduce the variance of the policy gradient estimate, making the learning process more stable.

The key difference between PPO and its combination with A2C2 is the computation of the *PPO objective function* (lines 15–16). This objective function is designed to penalize the model if the new policy deviates too much from the old policy. This is achieved by a clipping function that limits the ratio of the new and old policy probabilities. The hyperparameter ϵ controls the amount of change allowed in the policy update. This makes PPO more stable and reliable than vanilla policy gradient methods like A2C.

The update to the Critic network is also a form of *Temporal Difference (TD) learning*. The Critic estimates the expected return (or value) of each state, and these estimates are updated based on the TD error, which is the difference between the estimated value of a state and the estimated value of the next state, adjusted by the reward received. The TD learning update can be seen in lines 18 and 19, where $J(\phi)$ is the TD error used to update the parameters ϕ of the Critic network. So, while CARL incorporates elements of Adversarial Learning via the discriminator and policy gradient methods via the Actor, it also uses TD learning as part of its approach to learning a policy that imitates expert behavior.

Thus, CARL allows the agent to learn a policy that imitates the expert's behavior, while also optimizing the policy using the PPO algorithm. The discriminator network plays a crucial role in this process as it provides an additional reward signal

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446

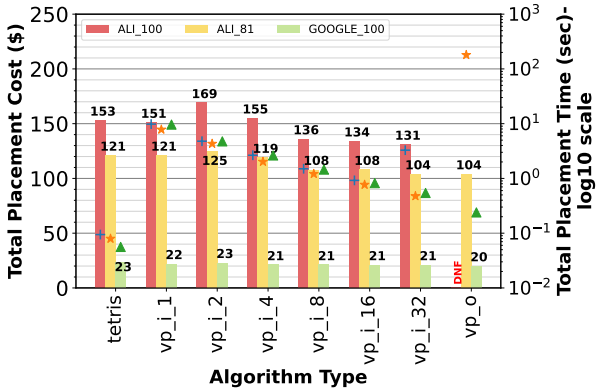


Figure 2: VM $ cost (bar, left Y axis) and decision-making time (marker, right Y axis) to place tasks from *Alibaba and Google traces* using *Tetris* and *VPSolver (VP)*.

that encourages the agent to imitate the expert.

3) Discussion: Adversarial training typically involves a single model, which is being made robust, and a method for generating adversarial examples for improving it. In CARL, we harness generative adversarial training from GAIL [17] to fit distributions of the states and the actions that define the behavior of the VPSolver expert. Though adversarial methods are sensitive to larger changes in data distribution, our careful design of the generator environment, and the proposed action clustering strategy help tackle the sensitivity to a large extent. When this is combined with the optimization-based PPO in the CARL generator (A2C2), the policy becomes more resilient. CARL is is made more effective since we have access to expert demonstrations, in our case, the output of VPSolver. We see this in our experiments where modest changes to the workload do not require re-training of the original model but large changes cause the quality to deteriorate.

CARL can also be generalized to scenarios where the number of VMs are fewer than the original VM count when training. This is done by setting resource vectors of the additional VMs to zero. By having a large enough number of VMs when training, as we do in our experiments, we avoid having to retrain the model for a more resource constrained VM pool. Also, we can handle future requirements such as placing containers in complementary VM availability regions for reliability, by action masking logic. Such flexibility helps address a wide range of requirements without fundamentally changing the training approach.

C. Contrasting VPSolver and Tetris as an Expert

We need a placement algorithm that acts as an expert during offline training of CARL. This algorithm needs to generate high-quality solutions, i.e., minimize the incremental VM costs paid after placing a set of container requests, based on the prior state of active and inactive VMs, and do so in a reasonable amount of time to generate sufficient trajectories for training. In our experiments, we use 1000–5000 training samples, depending on the workload.

We evaluate the performance of VPSolver and Tetris as two classic candidates to serve as the expert. There is a cost–benefit trade-off: VPSolver is expected to be slower but give higher quality solutions, while Tetris is faster but less efficient. *Tetris* places one container request at a time, as it arrives, within the available VMs, and operates in an online manner over the request stream. However, the *optimal VPSolver*, VP_o, is run in an offline manner as it requires *a priori* knowledge of the full request trace across time ("oracle") to provide the optimal lowest cost. Since this is not feasible in a streaming scenario, we also evaluate *incremental VPSolver*, where we group the incoming requests into batches of size $n = \{2, 4, 8, 16, 32\}$ items, and incremental run the solver on each batch ($VP_{i\_1} \ldots VP_{i\_32}$). While this does not yield a globally optimal solution, it makes locally optimal decisions for each batch based on prior state and, as we see, having larger batch sizes progressively offers lower costs; so it is semi-optimal.

We evaluate Tetris, VP_o and VP_i for placing 100 container requests each from the Alibaba [43] and Google [44] traces (ALI-100 and GOOGLE-100) onto $\approx$ 200 VMs (discussed later in § V-A). While we use 1000s of samples in the actual training of CARL, these 100 shown here help us make a judicious selection of the expert. We report the total VM cost in US$, which is a function of VM's price (Table I) and the duration it was active for. Alibaba's containers have longer lifespans compared to Google. Since a VM's active time is derived based on the longest execution time of all the tasks placed in that bin, Alibaba's VM costs are higher.

In Figure 2, we see that for both ALI-100 and GOOGLE-100, the VM costs mostly reduce for the solution provided by VP_i as the batch size increases from 1 to 32. This reduction is higher for ALI-100, reducing from \$151 to \$131, while it is more modest for GOOGLE-100, dropping from \$23 to \$21. For batch sizes of 8 or higher, VP_i is consistently cheaper than Tetris. Since Google's trace has lower resource needs compared to Alibaba (Figure 3), most tasks fit within a single VM. This offers many possible efficient packing choices and all strategies perform similarly.

Further, as the batch sizes grows, the VP_i solution approaches VP_o, which is the global optimal for an oracle. Here, ALI-100 was unable to provide an optimal placement solution even after executing VPSolver for 12h (Did Not Finish, DNF). The most it could finish in reasonable time was 81 tasks, which we report as ALI-81. VP_i with 16 and 32 batch sizes and VP_o are comparable in cost for ALI-81 and GOOGLE-100, making VP_{i_16} and VP_{i_32} good candidates for the expert, since they have visibility into a larger window of requests to optimize over. Tetris remains sub-optimal for ALI-81.

Tetris is fast and takes $<$ 100ms to solve for all placements (right Y axis in Figure 2). The total time to generate solutions for VP_i ranges from 1–10 seconds and, interestingly, reduces with the batch size since there are fewer batches to solve for. VP_i has a highly variable time since its optimal outcome depends on various factors of the large VM and container solution space. It ranges from 0.8s for GOOGLE-100 to 180s for ALI-81 to over 12h (DNF) for ALI-100. So VP_O is not evaluated further in our experiments.

Based on these micro-benchmarks, we choose VP_{i_16} as the expert for training our learning models, and as the default in future experiments. This balances the solution quality and

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446

time to generate the solution for training the RL models. For batches with more than 16 requests that arrive in a unit-time, we split them into sets of 16 and generate the placement solution incrementally for each.

V. Experiments

We comprehensively evaluate CARL for diverse workloads that demonstrate the robustness of our method. We also convincingly show its comparative benefits over two classic techniques, *Tetris* [4] and *incremental VPSolver* [20], and two state-of-the-art (SOTA) learning techniques, the *clustering-enhanced A2C2* [16] and using *Behavior Cloning (BC)* [21].

We next describe the comprehensive training and evaluation workloads derived from real-world traces of Alibaba [43] and Google [44] with up to $10k$ requests placed on up to 8000s VMs, discuss our training and evaluation harnesses, and report comparative metrics for the placement strategies. Conducting such large scale experiments with a large number of requests and VMs using real hardware resources is prohibitively costly.

A. Workloads

There are few large-scale task and container traces available publicly for clouds. Among these, the workload traces from Google and Alibaba are the most relevant to us and offer diverse characteristics. As described below, we use these to derive generic container request traces whose resource requirements and arrival patterns represent enterprise workloads.

1) Real-world Traces: The *Google cluster data v2.1* [44] has 29 days of trace data from May 2011 on the placement requests and resource usage of 23.25M tasks (which we treat as containers) on 12,500 machines (VMs) in a Borg cluster. We use the `Task_Resource_Usage_Table` that provides information about the CPU, memory and disk usage recorded at every 600s for each unique task request, given by the combination of `job_id` and `task_id`.

The *Alibaba Microservice Trace v2021* [43] reports the resource utilization of 96.45K containers used to host stateless micro-services on a production cluster with over 10,000 bare-metal servers (which we treat as VMs) managed using Kubernetes, over a 12-hour period. The resource usage of each container (micro-service instance) is recorded every 30 s in the `MS_Metrics_Table`, with a timestamp and the normalized utilization of CPU and memory.

In both cases, we consider the first occurrence of the task in the trace as its arrival time for placement, its last occurrence as the end time, and the peak observed resource utilization as its resource request during placement. Since the placement problem is scoped to a series on task requests, we omit any notion of groupings (e.g., jobs, workflows) formed from these.

For each request, we scale the normalized 0.0–1.0 value for a resource dimension to an actual resource value. For Google, we scale its CPU by 8 cores, memory by 32 GB and disk by 1024 GB, and assume a network usage (not part of the trace) proportional to the CPU, $\frac{CPU}{10}$ Gbps. For Alibaba, we scale the CPU by 8 cores and memory by 16 GB. Since disk and network usage are absent, we extrapolate them as CPU $\times$ 30 GB as disk usage and the network usage as 1, 2, 16 or 20 Gbps for CPUs of 2, 4, 8 and 16, respectively. These are based on nominal expectations of observing the distribution of such tasks, and helps map real numbers to the Integer space, required by the ILP-based VPSolver.

We **sample** a subset of tasks from both traces for our experiments. Such sampling (rather than just a *replay* of the traces) helps avoid over-fitting to individual patterns, and instead learns the overall distributions of the resource requests and inter-arrival times. It also helps create multiple workload instances by sampling the distributions for a robust evaluation.

The traces are *diverse* and offer contrasting choices on **resource requests**. From Figs. 3a and 3e, the median task duration in the original Google trace (ORIG) is 8.3 mins while the peak is about 686 h ($\approx$ 29 days). For Alibaba (ORIG), the median duration of a request is much higher at 3.8 h but the peak is limited to 11.9 h. Google also has many lightweight container requests compared to Alibaba. The median denormalized CPUs for Google containers is just 1 compared to 3 for Alibaba (Figs 3b, 3f), while the median denormalized memory is also just 1 GB for Google as opposed to 11 GB for Alibaba (Figs 3c, 3g). The ratios between the normalized CPU to memory for both are in Fig. 3d and 3h.

The probability distribution of the **inter-arrival times** between requests for these two traces is shown in Fig. 4. This determines the number of requests that arrive for placement decisions at any point in time. Over 95% of the requests for Alibaba are at timestep 0, likely because they are already present in the cluster when the trace collection begins. We omit timestep 0 from the inter-arrival times, leaving us with 99 container requests, which are shown in the CDF in Fig.4b (ORIG) in $1000\times$ seconds. While 75% of requests fall within 60 secs, some go up to 8000 secs. A similar CDF for Google (Fig. 4a, $100\times$ seconds) shows 94% of all requests with an inter-request duration of 0 secs. Sampling from this captures the inter-arrival periods of the container requests and helps generate the request arrival trace for the workloads below. This, in turn, helps form the set of requests for placement in each interval of scheduling, taken to be 1 sec windows.

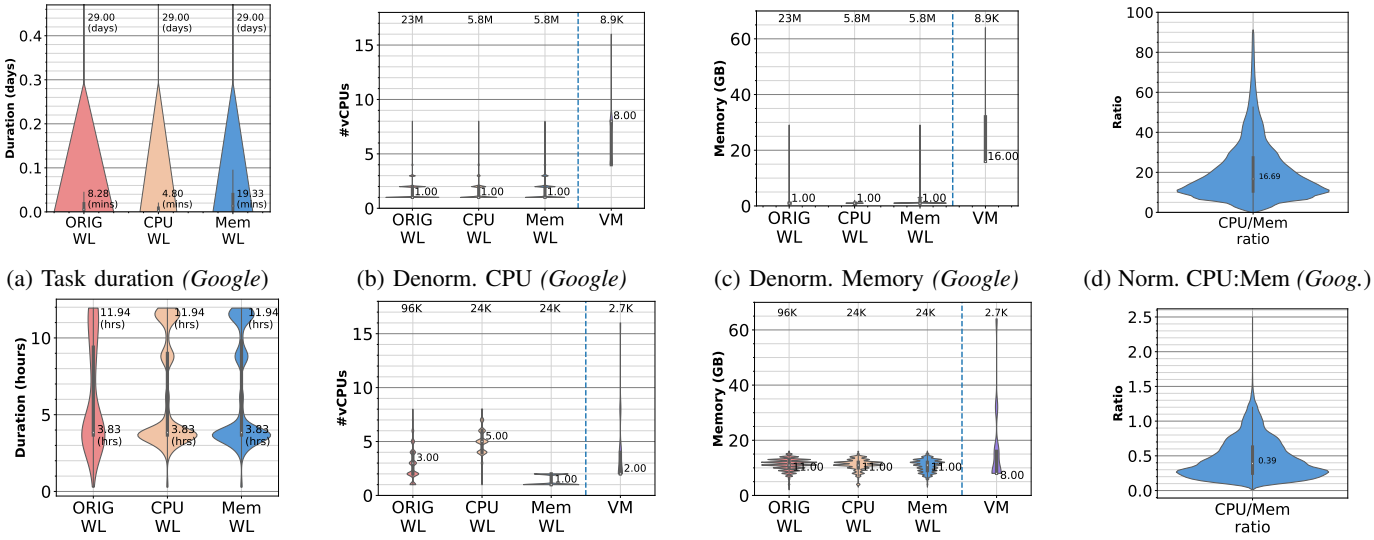
2) Creating Diverse Workloads from Traces: We derive workloads for training and validation by sampling requests from these original Google and Alibaba trace distributions. There are two parts to each workload: (1) The *resource needs* and *duration* of the requests, and (2) The *inter-arrival time* between requests, which decides the request-count in each placement batch. We next describe their sampling strategies.

We create one **training workload** and multiple **validation workloads**. We only use a small training workload with few requests to train the learning models, one each for Google and Alibaba. This keeps the training overhead low. The validation workloads are typically $2\times$ longer than the training ones. These models are then used to evaluate the placement accuracy for different validation workloads. We also *intentionally skew* some of the distributions to evaluate the robustness of the data-driven techniques to changes in the distribution and their generalizability under diverse conditions.

For each task in a workload, we sample and select a task from the relevant trace and adopt its resource vector. We separately sample the duration distribution to decide its lifetime.

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

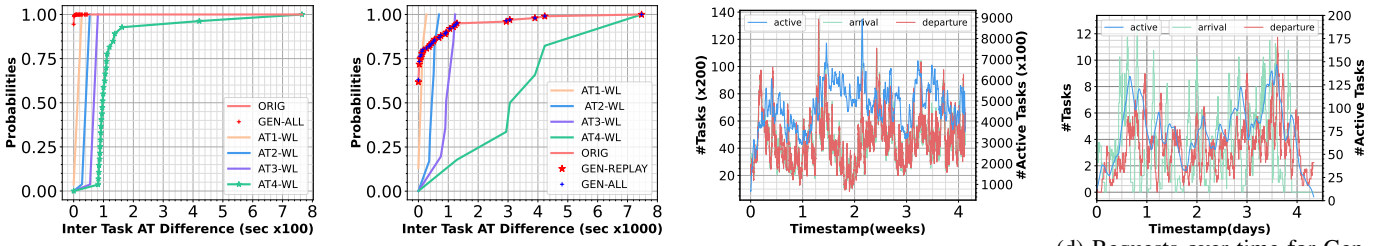
This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446



(a) Task duration *(Google)* (b) Denorm. CPU *(Google)* (c) Denorm. Memory *(Google)* (d) Norm. CPU:Mem *(Goog.)*

(e) Task duration *(Alibaba)* (f) Denorm. CPU *(Alibaba)* (g) Denorm. Memory *(Alibaba)* (h) Norm. CPU:Mem *(Alib.)*

Figure 3: Resource distribution and duration of requests for different workloads from Google (top) and Alibaba (bottom) traces



(a) CDF of inter-task arrival times *(Google)* (b) CDF of inter-task arrival times *(Alibaba)* (c) Requests over time for Gen-WL *(Google)* (d) Requests over time for Gen-WL *(Alibaba)*

Figure 4: Inter-arrival time between requests and the arrival and departure timestamps

We then sample the arrival time distribution to determine the timestamp between successive tasks in that workload.

a) Training Workload (Train-WL): We uniformly sample 5,000 requests from Google trace out of ≈ 23M and 1000 requests from Alibaba trace from ≈ 96K to serve as our training workloads. For Alibaba, given the small number of 99 inter-arrival time requests due to dropping timestep 0, we just replay the arrival rate of the original trace.

b) General Purpose Workload (Gen-WL): We uniformly sample 10,000 requests for Google out of ≈ 23M and 1000 requests for Alibaba from ≈ 96K for the general purpose validation workloads. These mimic the resource request behaviour of the original trace, which we treat as a generic workload. They also match the distribution of the training workload.

c) CPU and Memory Intensive Workload (CPU-WL, Mem-WL): Enterprises may have applications that are compute-heavy and require more CPU resources than memory or alternately memory-heavy. We capture such skewed workloads by sampling container requests from the first (Mem-WL) or fourth quartile (CPU-WL) of the normalized *CPU:Memory* distribution of the two traces (Figs. 3d and 3h). We sample 10k requests for each workload from 5.8M/24k requests from Google/Alibaba present in each quartile. As intended, their resource distributions diverge from Train-WL (Fig. 3).

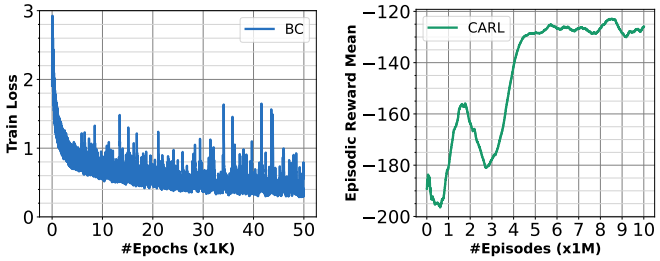
d) Arrival-time Skewed Workloads (AT1-WL — AT4-WL): Lastly, we design four workloads with skewed inter-arrival time distributions. This determines the expected number of requests in each 1 sec arrival window. For Gen-WL, we have a median/maximum of 12/124 requests per window for Google and 2/15 for Alibaba. We skew this by sampling the inter-arrival task times from the four quartiles of the original trace distribution to give us workloads AT1-WL, AT2-WL, AT3-WL and AT4-WL that correspond to the Q1–Q4 quartiles. E.g., Q1 of Google has inter-arrival times between 0–27 secs, while Q4 ranges between 82–765 secs, and Q4 for Alibaba spans 1290–7470 secs. The resource vectors and durations will continue to match the general workload. For each quartile, we sample 10k and 1k requests for Google and Alibaba.

e) Mixed Workloads (Mix-WL): This workload's resource mix changes across time, having characteristics of the General Purpose, CPU Intensive and Memory Intensive Workloads at different phases of time. This captures realistic workloads that may have periods of CPU or memory intensive requests that may arrive in the midst of generic workload requests. Specifically, use the Google Trace to sample $10k$ requests, with the first $5k$ taken from Gen-WL, next $3k$ from CPU-WL and the last $2k$ from Mem-WL.

3) VM Capacities: Our VM sizes are scaled from the Google and Alibaba traces, and their prices based on those charged by AWS public cloud for matching EC2 VM sizes [42]. The Google trace reports normalized CPU and memory capacities for the machines in their cluster. We use the 4 most common configurations that resemble general-purpose machines as our VM sizes. We denormalize by scaling

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446



(a) BC loss over epochs (b) CARL reward over episodes

Figure 5: Convergence of BC and CARL training

Table I: VM sizes used for Google and Alibaba workloads. Memory & Disk in GB, NW B/W in Gbps and Cost in US$/h.

VM	VMs for Google Workloads					VMs for Alibaba Workloads				
Size	CPU	Mem	Disk	NW	Cost	CPU	Mem	Disk	NW	Cost
S	4	16	120	10	0.186	2	8	60	12	0.113
M	8	16	240	10	0.204	4	16	120	12	0.192
L	8	32	240	10	0.326	8	32	240	20	0.326
XL	16	64	480	12	0.653	16	64	480	20	0.653

the CPU by 16 cores, memory by 64 GB, and set the disk capacity as 30×CPU GB and network bandwidth between 10–20 Gbps. We pick the VM price that matches the closest general-purpose EC2 VM size. Since Alibaba does not provide details of their servers, we select four general-purpose EC2 VM types: `m5d.large`, `m5.xl`, `m7g.2xl` and `m7g.4xl`. As is current practise by public cloud providers, we follow a per-second billing strategy for each active VM.

The number of VMs of each type is selected to comfortably accommodate all tasks of a workload simultaneously on instances of a single VM type. This relaxed approach prevents overflows while not affecting the billing since it is based on the active VMs used rather than the notional VMs available for placement. This also helps the trained CARL model to generalize to situations with fewer VMs. These VM types and counts available for placement are shown in Tables I and II, and their resource distributions show in Figs. 3b, 3c, 3f and 3g.

B. Training and Validation

1) Training of CARL and SOTA Baselines: We train models for the Google and Alibaba cluster traces separately using the training workloads from above.

A2C2 is trained by the interaction of the agent with the environment, guided by the reward signal, where the agent tries to place each incoming request on the set of available VMs, post clustering. For *CARL* and *BC*, we first generate expert demonstrations from $VP_{i_{16}}$ on the training workload trace, with a maximum batch size of 16 requests. It takes between 70–160s to generate 1–10k trajectories as expert samples for Alibaba and Google workloads. For CARL, a replay buffer of size 2048 stores past demonstrations from an expert. When training the discriminator and policy network, batches of trajectories are sampled from this buffer to update the agent's understanding.

BC is trained for $50K$ epochs, while A2C2/CARL are trained for $10M$ episodes with Adam optimizer, learning rate of 3×10^{-4} and a batch size of 2000. These are sufficient to train them till convergence in both cases, as shown in Fig. 5, ensuring fairness. Tables III and III show the hyper-parameters used for A2C2 and CARL. The generator network in CARL uses the same hyper-parameters as A2C2.

Table II: # of training and testing samples for container requests, and # of VM bin of different sizes

Trace	# of Container Requests				# of Virtual Machines				
	Train	GenWL	CPUWL	MemWL	S	M	L	XL	Tot.
Google	5000	10000	10000	10000	3990	1995	1995	998	8978
Alibaba	1000	1000	1000	1000	1475	737	368	184	2764

Table III: Hyper-parameters used for A2C2 and CARL

A2C2 Parameters	Value
Critic-network hidden layer	64×128
Actor-network hidden layer	64×128
Activation function	ReLU
Optimizer	Adam
Learning rate (α)	0.0003
Batch size	2000
Discount factor (γ)	0.99
GAE (λ)	0.95
Clip range (ϵ)	0.2
Entropy coefficient (H)	0.0
Value function coefficient	0.5
Max gradient clipping	0.5
CARL Parameters	**Value**
Discriminator Network	$128 \times 64 \times 32$
Replay Buffer	2048

We use OpenAI Gym [45] to design the custom RL environment to learn the placement policy. The adversarial algorithms use an interface provided by Human Compatible AI [46]. The agent policy for CARL is trained using A2C2, which use modified implementations from OpenAI Stable Baselines [47]. All experiments are performed on a workstation with an Intel Core i9 11th Gen CPU with 8-cores@3.50 GHz, 128GB RAM and an RTX 3080Ti GPU. Training is done on the GPU for BC, A2C2 and CARL, and take 12.5h, 18.5h and 20h, respectively.

2) Validation: We develop a simulation harness based on SimPy [48] to evaluate the placement strategies on the different workloads. The simulator iteratively replays the container placement requests from each workload being evaluated. The placement decision from the simulator is used to update the mapping of containers to VMs and update the VM resource availability. The simulator also considers the container exits from the input workload. The state of the system, mappings, the VM costs and the decision-making duration are logged after every scheduling decision for plotting and analysis.

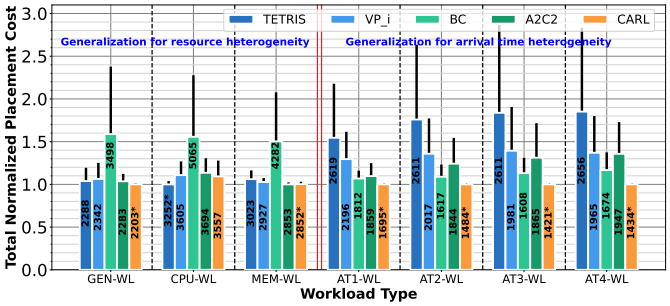
C. Results

We evaluate CARL and the baselines on their ability to minimize the *total US$ cost* paid for active VMs after placing the workload requests, and the *latency* for taking real-time placement decisions. We first discuss the General Workload (Gen-WL) and later examine their generalizability to skewed workloads that deviated from the training data.

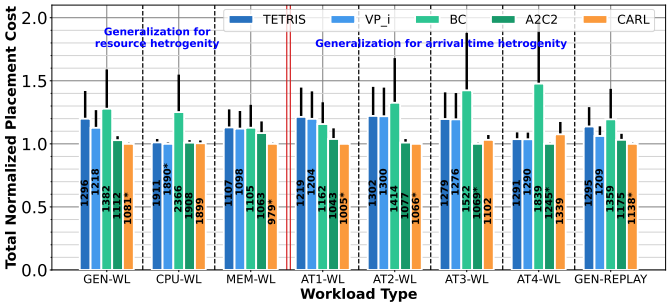
1) General Workload: Figures 6a and 6b show the **total normalized placement cost** for each strategy and each workload of the Google and Alibaba traces. For each workload type on the X axis, we create 5 workload instances through

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446

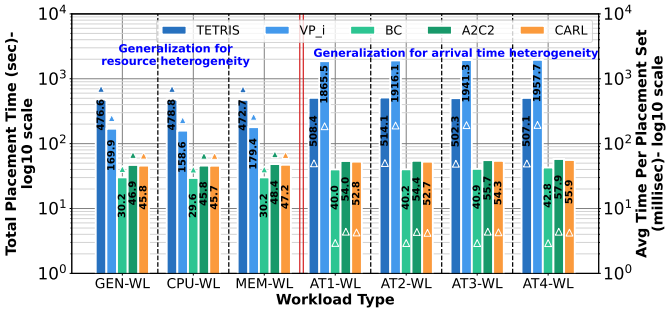


(a) Google Trace

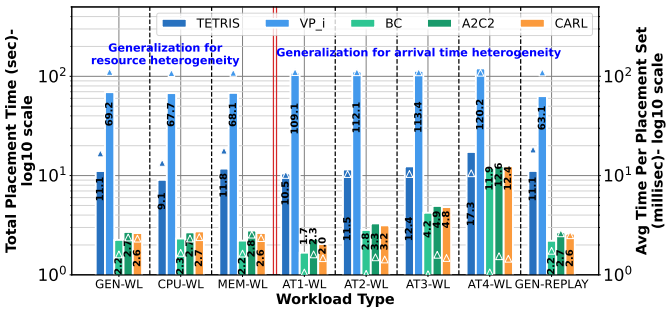


(b) Alibaba Trace

Figure 6: Total normalized placement costs for different workloads using scheduling strategies. Cheapest strategy is shown as 1.0. Labels on the bars indicate actual US$ cost.



(a) Google Trace



(b) Alibaba Trace

Figure 7: Total placement decision time (in seconds, log scale) for different workloads using different strategies

separate sampling and run the strategies on each instance for evaluation.

Within each workload, the strategy with the lowest US$ cost is assigned a normalized cost of 1.0 while we report the cost ratio for the other strategies (≥ 1.0) relative to this. The text labels on the bars show the actual total US$ cost for placing of all requests in the workload, averaged across instances. An '*' indicates that is has the lowest cost. An error bar shows the variability in costs across the instances.

When we consider the first cluster of bars for *Gen-WL*, we see that CARL (last bar, orange) consistently gives the lowest cost for placement, followed by A2C2. The other strategies are costlier than CARL by 3.6–58.8% for Google and 2.9–27.9% for Alibaba. A2C2, which is a building-block for CARL, costs 3–4% more than CARL – $80 costlier for Google, while BC is 59% costlier, at $1295, and also has a high variability.

Tetris and VP_i are comparable, with the former cheaper for Google and the latter for Alibaba. Incremental VPSolver, which is the expert teacher for CARL and BC, is itself worse than CARL by 6.3–12.6% since it only has access to the current batch of 16 requests for local decisions over the current VM states. Further, the small resource needs of containers relative to the VM sizes (Figures 3b, 3c) influences the packing efficiency when done incrementally.

CARL has learnt not just from VP_i's earlier solutions but is further tuned by the discriminator based on environment interactions with A2C2 to learn a better reward function – helping it outperform even VP_i. BC, on the other hand, is trained using the same expert but is limited to performing only as good as situations previously seen and solved by VP_i.

Further, CARL uses *Temporal Difference (TD) learning*. The Critic in CARL, estimates the expected return (or value) of each state, which are then updated based on the TD error – the difference between the estimated value of a state and the estimated value of the next state, adjusted by the reward received. So, while CARL incorporates Adversarial Learning using the discriminator, and policy gradient methods using the Actor, it also uses TD learning to learning a policy that imitates and surpasses the expert behavior.This establishes the superior design of CARL over other SOTA baselines.

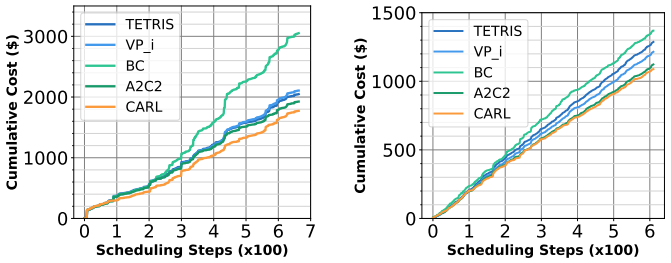
Besides the overall costs for the entire workload, it is also useful to see the **cumulative costs** at different progressions of the trace. This is shown in Figure 8 for Gen-WL after each step of placement, for a representative workload instance. The requests span 664 scheduling steps for Google's 10k container requests, and 611 steps for Alibaba's 1k requests. Beyond the initial 100 steps, CARL consistently out-performs others and is the cheapest, followed by A2C2. This confirms the *sustained performance* of our strategy over the baselines.

Figure 7 shows the **total time to make placement decisions** (in seconds, left Y axis, log scale) by the different strategies for the entire workload, and the **average time per batch** (in ms, right Y axis). CARL, A2C2 and BC are the fastest across all workloads, taking only 3–47 secs to place 1k requests of Alibaba and 10k requests of Google for Gen-WL. Since Google has 10× more requests than Alibaba, it is slower. Per batch times (denoted by markers), follows a similar trend with average placement being within 3.5 ms (Alibaba) to 65 ms (Google) for RL methods. Specifically, even for the largest placement batch of 124 requests CARL takes just 65 ms, translating to ≈ 1900 requests/sec.

These confirm that CARL is *fast enough for online placement decisions* even with 1000 of requests per second and over 10,000s of VMs, and can give a *cost-effective solution*

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446



(a) Google Trace (b) Alibaba Trace

Figure 8: Cummulative placement cost for GEN-WL over time for scheduling strategies (lower is better)

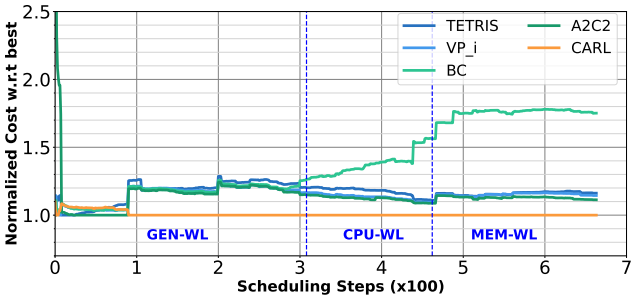


Figure 9: Normalized cost relative to best strategy in a step (1.0), for all strategies as workloads vary over time.

compared to both classic and ML baselines. Next, we expand this claim to the generalizabiluty of CARL with variations in the input workload compared to what it was trained on.

2) Adaptation to Diverse Workloads: While the long-term resource distribution for a data center may be represented by the Train-WL/Gen-WL, there may be transient conditions where container requests that are CPU-heavy or memory-heavy may arrive. Here, we examine the resiliency of the RL models trained on the Train-WL for placement of workloads whose resource distributions are much different. This evaluates the ability of a single model to still offer good placement decisions in such situations. In Figure 6 we report the normalized placement cost when using the original RL models for CARL, A2C2 and BC to place the CPU-WL and Mem-WL workloads, where all container requests are either CPU-intensive (Q4 of the CPU:Mem ratio) or memory-intensive (Q1).

All strategies except BC accrue costs that are within 1–8% of each other for the CPU-WL and Mem-WL of Google, and within 1% of each other for the CPU-WL of Alibaba. For CPU-WL on Google, Tetris is the best and has 8% lower costs than CARL, A2C2 and VP_i. CARL gives the lowest cost for the two other workloads. It is marginally better than others by $0.04\% - 5\%$ for Google MEM-WL and, much better by $7.9 - 11.6\%$, for Alibaba MEM-WL.

BC performs poorly across all four resource-skewed WLs, e.g., costing 57.6% and 24.6% more than CARL for the CPU-WLs of Google and Alibaba. This is again attributed to BC being unable to handle situations that are not present in the expert training, which is more so with these workloads. This reiterates the fact that using imitation learning in a generic manner using an expert does not work for solving the placement problem, and improving it with adversarial learning in CARL allows it to perform well, even for resource distribution skews. The decision-making time for these workloads, shown in Figure 7, are also comparable to Gen-WL since they have a similar number of requests.

3) Adaptation to Workload Variability Across Time: We further examine the impact of mixed workloads when their behavior changes in the midst of an ongoing execution, as may be expected in the real-world. We place requests from the Mix-WL (50% Gen-WL requests, then 30% CPU-WL requests, and 20% Mem-WL requests) using all scheduling strategies, trained on the original Gen-WL. We report a representative result from 5 runs in Fig. 9. The plot shows the normalized cost for each strategy at each step, relative to the best performing one at that step, which is assigned a relative cost of 1.0.

CARL has a least relative cost of 1.0 at most steps, other than the start when it is $\approx 10\%$ over the best strategy. Across the 5 runs, CARL offers the lowest cost for 90.72% of the placements. This continues to demonstrate the robustness and generalizability of CARL in placement of the mixed workload as well, out-performing the other strategies. The varying resource requirements lead to a more complex decision-making process for the scheduling algorithms and we see that all strategies (including CARL occasionally in the other runs) show a change in behavior when a new phase of workload is initiated. However, CARL is able to adapt the best and returns a more efficient placement. This confirms that CARL does not over-fit to just the general workload it was trained on but has learned the placement characteristics for a request based on its resource needs and the current resource state of the system.

4) Adaptation to Arrival Time Variation: Lastly, we explore the effect of heterogeneity in the inter-arrival time on the placement quality. These have a similar resource distribution but different arrival rates compared to the workload the models were trained upon. The right side of Figures 6 shows the total normalized placement cost for these skewed workloads, AT1-WL – AT4-WL, from the four quartiles of the inter-arrival time distribution. For Alibaba alone, which has a short trace, we include another workload, *Gen-Replay*, which replays the requests of the original trace to mimic the behaviour of recurrent arrival patterns in some enterprise workloads.

The robustness of CARL over other baselines is much more evident with these skewed workloads (Figure 6a). CARL performs the best for the four Google workloads, with other strategies reporting 6.9–85.2% higher costs. Even A2C2, which usually is marginally worse than CARL, does much more poorly here, e.g., 36% and $513 costlier for AT4-WL. For the Alibaba workloads with arrival time skews, CARL performs the best, or within 3% of the best, for all but AT4-WL. Here, we see A2C2 stays competitive alongside CARL as the first or second cheapest. As expected, Gen-Replay resembles Gen-WL and CARL again does the best.

The total placement times plotted in Figure 7 remains small for the RL methods but shows some higher timings compared to Gen-WL for Alibaba, and it grows with the quartile. With longer inter-arrival time in each of the four quartiles, there are more batches required for the same number of requests; so, the decision-making time grows. As a result, for Google, Tetris and VP_i exhibit much higher timings, reaching 500–1900s.

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446

VI. Discussion and Conclusions

In this paper, we have proposed a novel RL-based method, CARL, to take online placement decisions for container placement requests onto VM. It blends the actor-critic method with adversarial imitation learning, boot-strapped using an ILP solver as a teacher. In our rigorous evaluations with traces from Google and Alibaba, CARL gives more cost-effective placement decisions than other state-of-the-art RL baselines such as the enhanced A2C2 and BC, the Tetris heuristic, and also out-performs VPSolver, which served as its teacher.

The training samples CARL needs from the teacher is small, only 5000, and this takes just a few minutes, and its training time is modest. Importantly, CARL is more robust to changes in the workloads, compared to the one it is trained upon. This allows it to offer better or comparable costs than other techniques with variability in resource and arrival time distribution. CARL, and the other RL methods, are fast enough to place 1000s of requests per second.

Our evaluation using simulation is necessitated by the need for large online scale experiments, which is the problem space we tackle, but which is costly to execute on real hardware. While this assumes ideal behavior of containers and VMs, the resource sandboxing offered by containers, microVMs and VMs that are extensively used by cloud service providers is expected to mitigate this impact and serves as a best effort. Also, standard techniques such as maintaining some warm VMs to hide startup delays and using placing VMs in different failure domains can be layered above our solution. Examining interference between colocated containers is a non-goal, as is co-scheduling of related containers based on their interaction patterns. These are possible avenues of future extensions.

Other future research include modeling variable behavior of certain container types during placement. Further, there is heightened interest in federated learning from traces collected from across different enterprise regions if regulatory reasons prevent centrally sharing them.

References

[1] Y. Gan *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *ACM ASPLOS*, 2019.

[2] Cloudflare, "Cloudflare workers scale too well and broke our infrastructure, so we are rebuilding it on workers," https://blog.cloudflare.com/devcycle-customer-story/, 2022.

[3] Machines & Cloud, "Triumphs in kubernetes deployment: Inspiring case studies," https://machinesandcloud.com/successful-kubernetes-case-studies/, 2023.

[4] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 455–466, 2014.

[5] F. Brandão, "Cutting & Packing Problems: General Arc-flow Formulation with Graph Compression," Ph.D. dissertation, Faculdade de Ciências da Universidade do Porto, Portugal, 2017.

[6] B. Speitkamp and M. Bichler, "A mathematical programming approach for server consolidation problems in virtualized data centers," *IEEE Transactions on Services Computing*, vol. 3, no. 4, pp. 266–278, 2010.

[7] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," Microsoft Research, Tech. Rep., January 2011.

[8] M. Adhikari, T. Amgoth, and S. N. Srirama, "A survey on scheduling strategies for workflows in cloud environment and emerging trends," *ACM Comput. Surv.*, vol. 52, no. 4, aug 2019.

[9] H. I. Christensen, A. Khan, S. Pokutta, and P. Tetali, "Approximation and online algorithms for multidimensional bin packing: A survey," *Computer Science Review*, vol. 24, pp. 63–79, 2017.

[10] Z. Zhong, M. Xu, M. A. Rodriguez, C. Xu, and R. Buyya, "Machine learning-based orchestration of containers: A taxonomy and future directions," *ACM Comput. Surv.*, vol. 54, no. 10s, 2022.

[11] L. Gao, Y. Chen, and B. Tang, "Service function chain placement in cloud data center networks: A cooperative multi-agent reinforcement learning approach," in *Game Theory for Networks*, F. Fang and F. Shu, Eds. Springer Nature Switzerland, 2022.

[12] O. Hadary *et al.*, "Protean: VM Allocation Service at Scale," in *USENIX OSDI*, 2020.

[13] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *ACM EuroSys*, 2015.

[14] F. Cheng *et al.*, "Cost-aware job scheduling for cloud instances using deep reinforcement learning," *Cluster Computing*, vol. 25, no. 1, p. 619–631, feb 2022.

[15] N. Liu *et al.*, "A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning," in *IEEE ICDCS*, 2017.

[16] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz, "Reinforcement learning through asynchronous advantage actor-critic on a gpu," *arXiv*, no. arXiv:1611.06256, 2016.

[17] J. Ho and S. Ermon, "Generative adversarial imitation learning," *Advances in neural information processing systems*, vol. 29, 2016.

[18] Z. A. Mann, "Allocation of virtual machines in cloud data centers—a survey of problem models and optimization algorithms," *ACM Comput. Surv.*, vol. 48, no. 1, aug 2015.

[19] Z.-H. Zhan, X.-F. Liu, Y.-J. Gong, J. Zhang, H. S.-H. Chung, and Y. Li, "Cloud computing resource scheduling and a survey of its evolutionary approaches," *ACM Comput. Surv.*, vol. 47, no. 4, 2015.

[20] F. Brandão and J. P. Pedroso, "Bin packing and related problems: General arc-flow formulation with graph compression," *Comput. Oper. Res.*, vol. 69, pp. 56–67, 2016.

[21] K. Kang, D. Ding, H. Xie, L. Zhao, Y. Li, and Y. Xie, "Imitation learning enabled fast and adaptive task scheduling in cloud," *Future Generation Computer Systems (FGCS)*, 2024.

[22] P. Varshney and Y. Simmhan, "Characterizing application scheduling on edge, fog, and cloud computing resources," *Software: Practice and Experience*, vol. 50, no. 5, pp. 558–595, 2020.

[23] D. Bartók and Z. Á. Mann, "A branch-and-bound approach to virtual machine placement," in *HPI Operating the Cloud Symp.*, 2015.

[24] A. Anand, J. Lakshmi, and S. Nandy, "Virtual machine placement optimization supporting performance slas," in *IEEE International Conference on Cloud Computing Technology and Science*, 2013.

[25] M. Delorme, M. Iori, and S. Martello, "Bin packing and cutting stock problems," *Euro. J. of Oper. Res.*, 2016.

[26] O. Gurobi, "Gurobi optimizer," *http://www. gurobi. com/*, 2017.

[27] X. S. He, X. H. Sun, and G. Von Laszewski, "Qos guided min-min heuristic for grid task scheduling," *J. of Computer Science and Technology*, 2003.

[28] M. C. Silva Filho, C. C. Monteiro, P. R. Inácio, and M. M. Freire, "Approaches for optimizing virtual machine placement and migration in cloud environments: A survey," *JPDC*, vol. 111, pp. 222–250, 2018.

[29] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *NSDI*, 2011.

[30] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "A framework and algorithm for energy efficient container consolidation in cloud data centers," in *IEEE International conference on data science and data intensive systems*, 2015.

[31] A. Gupta, L. V. Kalé, D. Milojicic, P. Faraboschi, and S. M. Balle, "Hpc-aware vm placement in infrastructure clouds," in *IEEE International Conference on Cloud Engineering (IC2E)*, 2013.

[32] A. I. Orhean, F. Pop, and I. Raicu, "New Scheduling Approach using RL for Heterogeneous Distributed Systems," *JPDC*, vol. 117, pp. 292–302, 2018.

[33] R. Sridhar, M. Chandrasekaran, C. Sriramya, and T. Page, "Optimization of heterogeneous bin packing using adaptive genetic algorithm," in *IOP conference series: materials science and engineering*, vol. 183, no. 1. IOP Publishing, 2017, p. 012026.

[34] Z. Yang, P. Nguyen, H. Jin, and K. Nahrstedt, "Miras: Model-based reinforcement learning for microservice resource allocation over scientific workflows," in *IEEE ICDCS*, 2019, pp. 122–132.

[35] D. Zhang, D. Dai, Y. He, F. S. Bao, and B. Xie, "RLScheduler: An automated HPC batch job scheduler using reinforcement learning," in *ACM/IEEE SC*, 2020.

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3528446

[36] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv*, no. arXiv:1707.06347, 2017.
[37] Q. Wang, H. Zhang, C. Qu, Y. Shen, X. Liu, and J. Li, "RLSchert: An HPC Job Scheduler Using Deep Reinforcement Learning and Remaining Time Prediction," *Applied Sciences*, vol. 11, no. 20, 2021.
[38] W. Guo *et al.*, "Cloud resource scheduling with deep reinforcement learning and imitation learning," *IEEE IoT Journal*, vol. 8, no. 5, 2021.
[39] S. Ross and D. Bagnell, "Efficient reductions for imitation learning," in *Intl Conf on Artificial Intelligence and Statistics (AISTATS)*, 2010.
[40] I. Goodfellow *et al.*, "Generative adversarial networks," *CACM*, vol. 63, no. 11, pp. 139–144, 2020.
[41] Y. Pan, Y. Chen, and F. Lin, "Adjustable robust reinforcement learning for online 3d bin packing," *arXiv*, no. arXiv:2310.04323, 2023.
[42] "Amazon ec2 on-demand pricing," 2023, https://aws.amazon.com/ec2/pricing/on-demand/.
[43] S. Luo *et al.*, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *ACM SoCC*, 2021.
[44] J. Wilkes and C. Reiss, "Google cluster data," https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md, May 2011.
[45] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv*, no. arXiv:1606.01540, 2016.
[46] A. Gleave *et al.*, "imitation: Clean imitation learning implementations," *arXiv*, no. arXiv:2211.11972, 2022.
[47] A. Raffin *et al.*, "Stable-baselines3: Reliable reinforcement learning implementations," *JMLR*, vol. 22, no. 268, 2021.
[48] "Cloudsimpy: Simulation framework based on standard python for evaluating placement algorithms," 2023, https://github.com/FengcunLi/CloudSimPy.

Prathamesh Saraf Vinayak Prathamesh Saraf Vinayak is a member of the Cloud Systems Lab (CSL), and has an M.Tech. degree from the Department of Computational and Data Sciences (CDS), Indian Institute of Science, Bangalore. This article was part of this Masters' thesis. His work deals in reinforcement learning, optimization algorithms and software engineering. He hails from Sangli, Maharashtra.

Saswat Subhajyoti Mallick Saswat Subhajyoti Mallick is presently a researcher at the Robotics Institute, Carnegie Mellon University. His research deals in model diagnosis, neural rendering and 3D computer vision. He was a Project Associate in the Cloud Systems Lab (CSL), Indian Institute of Science, when this article was prepared.

Lakshmi Jagarlamudi Lakshmi J. is a Chief Research Scientist at the Supercomputer Education and Research Centre (SERC), Indian Institute of Science, Bangalore, for more than 27 years and heads the Cloud Systems Lab (CSL). Her research explores systems software and architectures for virtualized and distributed systems for cloud computing, including Quality of Service (QoS), and defining and meeting user expectation from clouds resources. She also manages and supports the HPC data center at SERC. She has a Ph.D. from the Indian Institute of Science. She is a recipient of Dr. A P J Abdul Kalam HPC Award in the year 2020. She is a senior member with IEEE and a member of ACM.

Anirban Chakraborty Anirban Chakraborty is an Associate Professor with the Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India. He received his Ph.D. degree in Electrical Engineering from the University of California, Riverside (UC Riverside) in 2014. He subsequently held research fellow positions with the National University of Singapore, and Nanyang Technological University, Singapore. After that, he was a Computer Vision Researcher with the Robert Bosch Research and Technology Centre at Bangalore, India. Anirban's research interest lies in Computer Vision, Machine Learning and their applications to diverse real-world problems. His recent research has focused on data-efficient and privacy-preserving deep learning, video surveillance problems and learning across modalities and domains. He is a member of IEEE.

Yogesh Simmhan Yogesh Simmhan is an Associate Professor in the Department of Computational and Data Sciences (CDS) and a Swarna Jayanti Fellow at the Indian Institute of Science, Bangalore. His research explores scalable software platforms, algorithms and applications on distributed systems. He is the recipient of the IEEE TCSC Award for Excellence in Scalable Computing (Mid Career Researcher) in 2020. He is an Associate Editor-in-Chief of the Journal of Parallel and Distributed Systems (JPDC), an Associate Editor of Future Generation Computing System (FGCS), and earlier served as an Associate Editor of IEEE Transactions on Cloud Computing. Yogesh has a Ph.D. in Computer Science from Indiana University, and was previously a Research Assistant Professor at the University of Southern California (USC), Los Angeles, and a Postdoc at Microsoft Research, San Francisco. He is a Distinguished Member of ACM, a Distinguished Contributor of the IEEE Computer Society and a Senior Member of IEEE.

Authorized licensed use limited to: University of Nantes. Downloaded on February 20,2025 at 00:52:27 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.