Towards VM Rescheduling Optimization Through Deep Reinforcement Learning

Xianzhong Ding*
University of California, Merced
xding5@ucmerced.edu

Yunkai Zhang*
University of California, Berkeley
yunkai_zhang@berkeley.edu

Binbin Chen
ByteDance
chenbinbin.1996@bytedance.com

Donghao Ying
University of California, Berkeley
donghaoy@berkeley.edu

Tieying Zhang†
ByteDance
tieying.zhang@bytedance.com

Jianjun Chen
ByteDance
jianjun.chen@bytedance.com

Lei Zhang
ByteDance
zhanglei.michael@bytedance.com

Alberto Cerpa
University of California, Merced
acerpa@ucmerced.edu

Wan Du†
University of California, Merced
wdu3@ucmerced.edu

Abstract

Modern industry-scale data centers need to manage a large number of virtual machines (VMs). Due to the continual creation and release of VMs, many small resource fragments are scattered across physical machines (PMs). To handle these fragments, data centers periodically reschedule some VMs to alternative PMs, a practice commonly referred to as VM rescheduling. Despite the increasing importance of VM rescheduling as data centers grow in size, the problem remains understudied. We first show that, unlike most combinatorial optimization tasks, the inference time of VM rescheduling algorithms significantly influences their performance, due to dynamic VM state changes during this period. This causes existing methods to scale poorly. Therefore, we develop a reinforcement learning system for VM rescheduling, VMR^{2}L, which incorporates a set of customized techniques, such as a two-stage framework that accommodates diverse constraints and workload conditions, a feature extraction module that captures relational information specific to rescheduling, as well as a risk-seeking evaluation enabling users to optimize the trade-off between latency and accuracy. We conduct extensive experiments with data from an industry-scale data center. Our results show that VMR^{2}L can achieve a performance comparable to the optimal solution but with a running time of seconds. Code[1][2] and datasets[3] are open-sourced.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Theory of computation → Reinforcement learning**; **Scheduling algorithms**; • **Software and its engineering → Virtual machines**; • **Applied computing → Data centers**.

Keywords: Cloud Computing, Virtual Machine Rescheduling, Reinforcement Learning, Resource Management

1 Introduction

Cloud service providers allow end-users to access computing resources, such as CPU and memory. They adopt resource virtualization to maximize hardware utilization, allocating Virtual Machines (VMs) [14, 60] with the requested resources to end-users. An industry-scale data center is typically organized into clusters, where each cluster has hundreds of Physical Machines (PMs), and each PM can host multiple VMs that run independently [20, 28]. However, if a PM already hosts several VMs and the remaining resources on the PM fail to fulfill an additional VM request, the resources leftover cannot be used are called fragments [56, 59]. To allocate the resources efficiently, a central server manages all VM requests on PMs by performing two tasks, scheduling and rescheduling, in order to achieve different resource utilization goals, such as minimizing the overall fragment rate (FR) or minimizing the number of migrations required to achieve a specific FR.

Fragment Rate. FR quantifies the ratio of unusable CPU resources to total available CPU resources across all PMs. Specifically, the numerator represents the total CPU resources that cannot be used to schedule a 16-core VM (i.e., CPU fragments that are too small or scattered to accommodate such a

*Both authors contributed equally to this research.
†Corresponding author.

EuroSys '25, March 30-April 3, 2025, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1196-1/2025/03.
https://doi.org/10.1145/3689031.3717476

[1]The majority of work was done during the internship at ByteDance.
[2]https://github.com/zhykoties/VMR2L_eurosys
[3]https://drive.google.com/drive/folders/1PfRo1cVwuhH30XhsE2Np3xqJn2GpX5qy

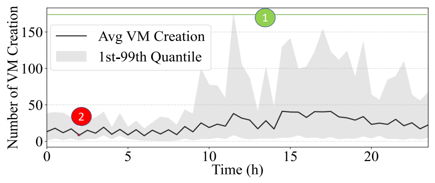


Figure 1. The number of VM arrivals and exits per minute. The green line indicates a continuous VMS process over 24 hours.

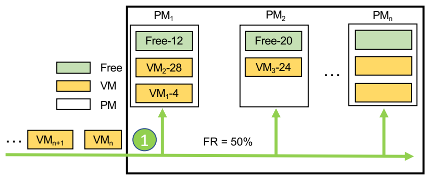


Figure 2. VMS process. The green number 1 denotes the VMS operation, selecting PMs for incoming VM requests.

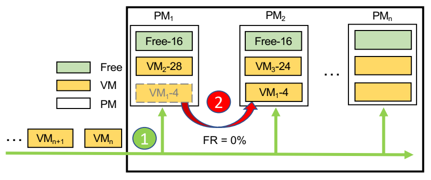


Figure 3. VMR process. The red number 2 highlights the off-peak period when VMR is typically performed.

VM). The denominator is the total available CPU resources across all PMs. This metric helps assess how efficiently the system utilizes its resources for large VM allocations.

VM Scheduling (VMS). When a new VM request arrives, VMS selects one PM from all available PMs that can accommodate the request. If a VM is not scheduled properly, it can directly affect the end user. Fig. 1 shows the distribution of VM changes (VMs arriving and exiting) per minute over 24 hours, averaged over a 30-day period from a cluster in our in-house data center. The y-axis represents the number of VMs changed (arrivals and exits) within each minute. To serve all users at all times, the VMS algorithm (green number 1) must handle the maximum number of VM changes, as indicated by the green line, which represents the continuous VM scheduling process throughout the day. The high queries per second (QPS) require VMS algorithms to have strict latency and stability, which deems only simple heuristic methods with short inference time feasible. In practice, ByteDance uses best-fit [27, 50], which sorts all PMs that meet the requirements of the current VM according to the amount of FR reduction before and after this VM is added, and chooses the PM with the largest reduction. However, such heuristic algorithms lead to many fragments scattered across PMs. Combined with the continual exiting of completed VMs, it leads to many fragments scattered across PMs. These have to be solved by VM rescheduling.

VM Rescheduling (VMR). Rescheduling is critical to optimize resource usage, which migrates VMs from their current PMs to new destination PMs. Unlike VMS which needs to run throughout the day, VMR is mostly performed during off-peak hours in early mornings[4] where there are fewer VM changes as indicated by the red dot in Fig. 1. Also, if a rescheduling action fails, VMs can simply stay on their original PMs without affecting the end-users. This allows the use of more advanced algorithms.

VMR can be efficiently performed using live migration, ensuring minimal downtime. As most data centers manage VMs with compute-storage separation (i.e., using cloud disks) [38], only the memory needs to be transferred. Specifically, we first copy the VM's memory state from the source PM to the destination PM while it continues running on the source PM. Changes to the VM's memory during this process, known as "dirty pages" are tracked and re-copied incrementally [14] until the remaining changes are small. At this point, the VM is briefly paused for a final synchronization. Since modern data centers use high-bandwidth networks for internal file transfers [62], this VMR process incurs a low overhead.

Note that rescheduling is primarily applied to clusters hosting VMs that use hardware virtualization, similar to Elastic Compute Cloud (EC2) environments. These VMs provide strong isolation and have a high startup cost, making them suitable for workloads running for extended periods, such as development machines [46]. Rescheduling is unnecessary for other short-lived tasks such as CI/CD or CronJob. They are managed in separate clusters via Kubernetes, which offers fast startup through operating system-level virtualization [36]. For system stability, rescheduling is typically restricted to the same cluster. A cluster typically involves no more than several hundred PMs, since i) it allows allocation of dedicated resources to different user groups, where specific configurations can be better optimized, and ii) each cluster can be monitored and managed independently, allowing one cluster to upgrade without affecting others [1]. A migration number limit (MNL) is set to control the number of VMs to migrate and is often chosen to be $2 \sim 3\%$ of all VMs.

Smaller VMs (e.g., proxy servers[5] or routine monitoring/testing) are easy to create using fragmented resources and have almost no risk of supply interruption. Conversely, many high-priority tasks that are directly consumer-facing require medium and large-sized VMs. Thus, our study focuses on the 16-core FR to meet the operational needs at ByteDance[6], where 16-core is the default VM type for development machines.

Benefits of VMR. Consider the FR in Fig. 2. PM_1 has 12 CPUs left and PM_2 has 20 CPUs left, but only PM_2 can host another 16-Core VM, and the remaining $12 + (20 - 16) = 16$ CPUs become fragments. The FR is therefore $16/(12 + 20) = 50\%$. In Fig. 3, VMR reassigns VM_1 from PM_1 to PM_2, leaving

[4]In less common cases, VMR is also performed if a high FR is observed that could potentially lead to insufficient resources for upcoming VM requests.

[5]A proxy server acts as an intermediary between a client requesting a resource and the server providing that resource.

[6]A simple extension allows our framework to accommodate FR defined based other X-Core VMs under different specifications, in the form of a weighted average of several VM types, or even a combination of CPU and memory fragments. See Section 5.5.

16 free CPUs on each PM, which is just enough to handle an additional 16-Core VM. The FR after VMR becomes 0%.

Note that while the VMR algorithm computes a solution, VMS is still handing new VM requests and completed VMs are also being deleted. The dynamic nature of VM states causes the computed VMR solution to no longer be optimal or even feasible[7]. Therefore, VMR also needs to be very efficient. In Section 2, we formulate VMR as a Mixed Integer Programming (MIP) problem and conduct an experiment to show that different from other MIP applications, VMR inference time must be under five seconds for the solution to remain competitive.

Most existing solutions either involve accelerating MIP solvers [48, 66] or rely entirely on heuristics [27]. However, the former still fails to meet the strict latency requirement, while the latter leads to suboptimal solutions. In this work, we develop VMR^{2}L, a deep Reinforcement Learning (RL) system for VM rescheduling. RL is a great fit for two reasons. First, while RL often suffers from poor sample complexity [37], VMR operates in a deterministic environment, meaning that, given the current state and action, the next state can be predicted exactly. This allows us to build a simulator that only requires the initial VM-PM mappings for training, without having to interact with a real data center, which *drastically lowers* the number of training samples required. Second, the generalization ability [45] of deep RL enables the agent to train offline and apply the learned policy directly in production *without retraining*. This is crucial in meeting the strict latency requirements for VMR. We summarize the contributions of this paper as follows:

- **RL for VM rescheduling.** We identify the unique characteristics of the rescheduling problem in terms of latency requirement and environmental uncertainties, which motivate its formulation as an RL problem.
- **Customized techniques for VM rescheduling.** We design i) a two-stage framework to flexibly accommodate different service constraints and address the exploration challenge, ii) a feature extraction module that scales to large data centers while capturing relational information specific to VMR, and iii) a risk-seeking evaluation pipeline that leverages the deterministic nature of VMR to offer a better trade-off between inference speed and solution quality.
- **A VMR^{2}L prototype and extensive evaluation.** We collect two real datasets and show that VMR^{2}L can generalize to different objectives, service constraints, as well as abnormal workloads at deployment time. Our code and datasets are released.

[7]A VM will not be rescheduled if it has exited or the destination PM no longer has enough resources or fails to meet other service constraints.

2 Motivation Experiment

2.1 Problem Formulation and Two Algorithms

In a data center cluster, let $\mathcal{V}, \mathcal{P}$ be the set of VMs and PMs, respectively. On the supply side, a PM $i \in \mathcal{P}$ is equipped with two Non-Uniform Memory Access (NUMA) nodes[8]. For PM i, NUMA j can provide $U_{i,j}$ CPU resources and $V_{i,j}$ memory resources. On the demand side, a VM $k \in \mathcal{V}$ requires u_k CPU resources and v_k memory resources and should be deployed on a single PM using $w_k \in \{1, 2\}$ NUMAs. w_k is the number of NUMAs required by VM k (1 for single-NUMA deployment, 2 for double-NUMA). After deploying several VMs on PM $i \in \mathcal{P}$, it remains $\tilde{U}_{i,j}$ spare CPU resources on NUMA j. We define **X-core fragment** of PM i as $\sum_j (\tilde{U}_{i,j} \% X)$, i.e., the remaining CPUs cannot be further utilized by additional X-core VMs.

Given M VMs that are initially assigned to N PMs, VMR reassigns a subset of deployed VMs and migrates them onto some new PMs. The *Migration Number Limit (MNL)* is a tunable parameter that defines the maximum number of VMs that can be migrated during each rescheduling task. By adjusting MNL, we can control the trade-off between performance improvements and migration overhead. We formulate VM rescheduling as an optimization problem that searches for the optimal reassignment of VMs (up to the specified MNL), with the goal of minimizing the total X-core fragments across all PMs:

$$\text{Minimize:} \qquad \sum_{i,j} \left(U_{i,j} - \sum_k \frac{x_{k,i,j} \cdot u_k}{w_k} - X y_{i,j} \right) \quad (1)$$

$$\text{Subject to:} \qquad \sum_k \frac{x_{k,i,j} \cdot u_k}{w_k} + X y_{i,j} \leq U_{i,j}, \quad (2)$$

$$\sum_k \frac{x_{k,i,j} \cdot v_k}{w_k} \leq V_{i,j}, \quad (3)$$

$$\sum_{i,j} x_{k,i,j} = w_k, \quad (4)$$

$$\sum_k (1 - x_{k,i_k,j_k}) \leq MNL, \quad (5)$$

$$x_{k,i,0} = x_{k,i,1}, \quad \forall k \in \{k | w_k = 2\}, \quad (6)$$

$$x_{k,i,j} \in \{0, 1\} \text{ and } y_{i,j} \in \mathbb{Z}. \quad (7)$$

Here, $\{x, y\}$ are the decision variables, where $x_{k,i,j}$ represents whether VM k is deployed to the NUMA j of PM i in the new assignment (0 for No, 1 for Yes), and $y_{i,j}$ represents the maximum number of X-core VMs can be deployed on NUMA j of PM i using the remaining CPU resources. The objective in Equation 1 is to minimize the total X-core fragments.

[8]A NUMA node is a subsystem within a PM that controls both memory and processing units. This architecture improves performance by allowing each node to access its local memory faster than memory on another node [5].

Table 1. VM types considered in the main experiments. Extra resource and service constraints are considered in Section 5.

VM Types	large	xlarge	2xlarge	4xlarge	8xlarge	16xlarge	22xlarge
Requested CPU	2	4	8	16	32	64	88
Requested Memory (GB)	4	8	16	32	64	128	176
Deploy NUMA	Single	Single	Single	Single	Double	Double	Double

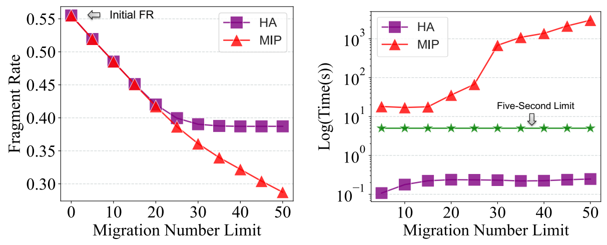


Figure 4. FR and inference time at different MNLs.

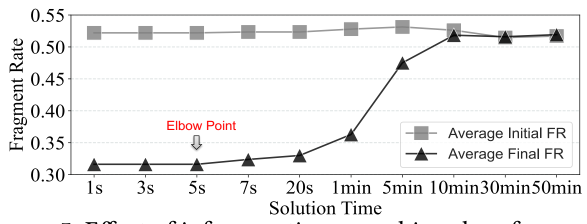


Figure 5. Effect of inference time on achieved performance.

Equation 2 and 3 enforce that the resource usage by VMs cannot exceed the total capacity of a PM. Equation 4 indicates that each VM must be deployed on exactly one PM. Equation 5, in which i_k and j_k are the initial PM id and NUMA id (0 for double-NUMA VMs) of VM k, means the total migration number should not exceed the limit. Lastly, Equation 6 forces VMs with double NUMAs to deploy both NUMAs on the same PM.

Note (1) each PM has two NUMAs; (2) w_k is a constant for each VM as determined by their types (Table 1). Thus, $\sum_{i,j} x_{k,i,j} = w_k$ (Equation 4) enforces that the actual NUMA allocation number of VM k matches the desired configuration. When $w_k = 1$, Equation 4 constraints VM k to be deployed on one NUMA of a PM; when $w_k = 2$, Equation 4 constraints VM k to be deployed on both NUMAs of a PM. Note that deploying VM k on two NUMAs of two different PMs (each PM hosting a NUMA) violates $x_{k,i,0} = x_{k,i,1}, \forall k \in \{k|w_k = 2\}$ (Equation 6). Because $w_k \neq 0$, it guarantees each VM is deployed.

Mixed Integer Programming (MIP) Solvers. The above optimization problem can be solved by an off-the-shelf MIP solver such as CPLEX [2] and Gurobi [3], which finds a near-optimal solution through branch & bound, cutting planes, etc. In our experiments, we use Gurobi.

Heuristic Algorithm (HA). To obtain a feasible solution within a short time frame, heuristic algorithms are often used in industry data centers [4]. They normally include two stages: filtering and scoring. In the filtering stage, we calculate the change in FR for each VM as if it is removed from its source PM, and only select the VM candidate that corresponds to the most drop in FR. In the scoring stage, we calculate the change in FR as if the selected VM is migrated to each of the eligible PMs. We then greedily assign the selected VM to the PM that leads to the largest drop in FR. The above two stages are repeated until the MNL is reached.

2.2 Experiment Results

We conduct experiments to quantify the performance of the above MIP and HA in terms of FR and inference time. We use a real dataset from a data center with 280 PMs and 2089 VMs. The detailed experiment settings can be found in Section 4.

Fig. 4 (left) depicts the FR performance of MIP and HA under different MNLs. MIP's FR is lower than HA's since it guarantees a near-optimal solution. Moreover, the FR gap between MIP and HA becomes larger as MNL increases, because HA only exploits the action that would lead to the most significant drop in FR when it migrates one VM. After migrating 25 VMs, the heuristic algorithm can no longer find any more VMs that can lower the FR. In Fig. 4 (right), we see that as the MNL increases from 25 to 50, the computation time of MIP grows exponentially, taking 1.78 minutes for 25 migrations and 50.55 minutes for 50 migrations. This poor time complexity is unacceptable in data centers where new VM requests continually come in, and the problem state is constantly changing.

To see how a near-optimal solution can result in a suboptimal achieved performance due to its poor inference time, we conduct an experiment on real traces from our in-house data center by selecting 200 random initial VM-PM mappings. For each mapping, we use Gurobi to compute a near-optimal solution to the MIP formulation of VMR, which takes 50.55 minutes. However, since VMs were dynamically arriving and exiting, most actions were no longer feasible and would fail to be deployed after 50 minutes. We then compute the final performance that could be achieved as if the near-optimal solution was instead returned in a shorter period of time, averaged over the 20 mappings. Fig. 5 shows that the solution remains near-optimal if it could be computed within five seconds, as highlighted by the "elbow point". However, FR reduction quickly diminishes once the inference time exceeds that. Therefore, we require all methods to be able to return a solution within a ***five-second limit*** for each mapping during inference (green line in Fig. 4 (right)).

Limitations of Current Methods. From the above experiment, we see that the primary issue of the MIP approach is its poor time complexity, which prevents it from scaling to industry-scale data centers with thousands of PMs and VMs. To reduce the execution time of MIP solvers, some current methods rely on estimating feasible solutions using proprietary heuristic methods and then using branch-and-cut techniques [48] to identify optimal solutions. The hand-tuned heuristics are based on human expertise to overcome the scalability challenge by pruning the search space. Yet, the heuristics have to be designed separately for different cluster conditions in every data center as there are no universal heuristics for all scenarios. For this reason, at ByteDance, we use Partitioned Optimization Problems (POP) [47] that randomly partition the rescheduling problem into several subproblems and apply MIP to each subproblem. However, as we show in Section 5.1, POP performs suboptimally under the five-second limit. Instead, we aim to design a solution that can match the performance of MIP, meet the latency requirements, and does not require manual feature engineering for different clusters.

3 Design of VMR^{2}L

3.1 VM Rescheduling as an RL Problem

Deep RL has demonstrated remarkable abilities in many domains [17, 57]. Without requiring pre-programmed heuristic rules from experts, deep RL *learns directly from data*, but its main drawback is the amount of training data required [7, 16]. Additionally, rescheduling MNL VMs simultaneously requires an action space of $O((M \cdot N)^{MNL})$, which is too large for the agent to learn effectively.

To address these challenges, we formulate the problem such that a VMR request starts an episode, which involves MNL steps. At each migration step, the action of the agent (VMR^{2}L) reschedules a single VM from its source PM to a new destination PM based on the current PM and VM status. Notably, the environment is deterministic – given the current state and action, we can exactly know the next state and the change in objective.

To this end, we design a simulator following the OpenAI Gym environments [11] that allows us to train VMR^{2}L offline — we collect training samples from the data center, where each sample is the status of all VMs and PMs when a VMR request is created. Each sample serves as the initial state of an episode. For each step in the episode, given the current state and the agent's action, the simulator computes i) the next state and ii) a reward based on the change in the rescheduling objective, without the need to interact with the actual data center. The agent in turn uses this reward signal to gradually improve the rescheduling policy. At deployment time, using neural networks as feature extractors allows VMR^{2}L to generalize to VM-PM mappings not encountered during training without retraining or finetuning.

Given the above design, we now formalize the state, action, and reward of the VMR^{2}L framework.

State Representation. At each step, the state serves as input to the agent, which is parameterized using neural networks. The state input contains two sets of features. The first set is the PM features, which contain four features for each of the two NUMAs of each PM, specifically the remaining CPU and memory resources, current FR, and fragment sizes. The second set is the 14 VM features, which include requested CPU and memory for each NUMA, fragment sizes, concatenated with the source PM features. If a single NUMA is requested, zeros are used as placeholders for the other NUMA. Each feature dimension is min-max normalized.

Action Representation. Given the state at each step, the agent outputs an action to migrate one VM. The action at each step can be represented as a 2-tuple (k, i), which means to reschedule a VM $k \in \mathcal{V}$ from its source PM to a destination PM $i \in \mathcal{P}$. Note that the source PM can be retrieved once we select k, so we do not include it in the action space.

Reward Representation. Reward represents the immediate evaluation of the benefits each migration step brings under the given state. The goal of VM rescheduling is to minimize the FR across all PMs. While we could return the FR of all PMs as a single final reward to the agent after finishing an entire episode, it corresponds to a form of sparse reward which is known to be difficult for training [53], as the agent cannot easily attribute the drop in FR to a certain migration step. Instead, we propose to generate dense rewards and use the change in fragment sizes on the source PM and the destination PM as an intermediate reward at each step. Since we focus on 16-core CPUs, the maximum change in fragment size on a single NUMA due to adding or removing a VM is −15 to 15. We normalize the reward by dividing it with a constant $c = 64$ so that its range is naturally scaled down to $[-\frac{15\times 4}{64}, \frac{15\times 4}{64}]$ [30]. We calculate the rescaled fragment size on both NUMAs by

$$S_i = \sum_{j=0}^{1} \left(\tilde{U}_{i,j} \% X \right) \div c, \tag{8}$$

and define reward as

$$R = (S_{\text{before, src}} - S_{\text{after, src}}) + (S_{\text{before, dest}} - S_{\text{after, dest}}), \tag{9}$$

where $S_{\text{before},\cdot}$ and $S_{\text{after},\cdot}$ are fragment changes before and after the selected VM leaves (enters) the source (destination) PM.

3.2 Two-Stage Framework

When the RL agent chooses to reschedule a VM k from its source PM to a destination PM i, PM i must have enough available CPU and memory to host VM k. In practical scenarios, we may also consider additional constraints to ensure service stability. For example, an application may require some VMs to be hosted across different PMs, which can be enforced in the form of a hard anti-affinity constraint.

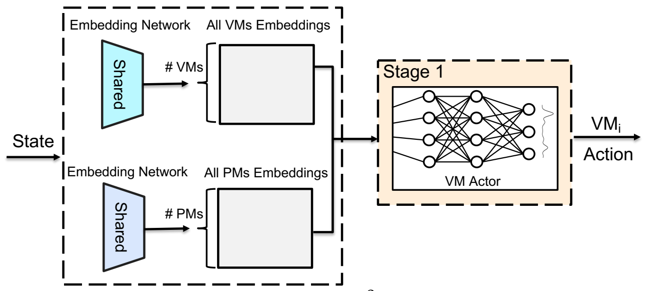


Figure 6. The first stage of VMR^{2}L processes all VMs and PMs via shared embedding networks, based on which the VM actor selects a VM to be rescheduled.

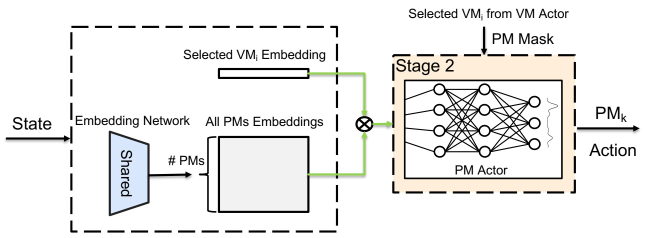


Figure 7. Once a candidate VM is selected by the VM actor, VMR^{2}L masks out all the PMs that cannot host the candidate VM. The PM actor only accesses the selected VM, and then selects a destination PM from the unmasked PMs.

Off-the-shelf RL models impose such hard constraints typically by invoking a heavy penalty if an illegal action is chosen, or by directly setting the probabilities for all illegal actions to be zero. As shown in Section 5.4, heavy penalties can result in gradient instability and lead to an inferior convergence rate. Furthermore, as the size of action space is $O(M \cdot N)$, masking all illegal actions is overly time-consuming.

To better accommodate a variety of constraints, we leverage the characteristics of VMR and design a two-stage framework that allows the action tuple to be generated sequentially. In Stage 1 (Fig. 6), the VM actor selects the VM candidate to be rescheduled. Once a candidate VM is selected, we can efficiently mask out all the PMs that cannot host the candidate VM. We then proceed to Stage 2 (Fig. 7), where the PM actor selects an appropriate destination PM from the remaining PMs. Such a framework has three benefits. First, it completely avoids illegal actions for various types of constraints and thus circumvents the necessity of heavy penalties. Second, it dedicates two separate networks to select the VM candidate and the destination PM, which simplifies the VMR task by decomposing the action tuple. Finally, when we select a VM to reschedule, a considerable portion of the PMs cannot meet its resource requirements. The proposed framework can avoid exploration on these PMs and thus mitigate the exploration challenge.

3.3 Feature Extraction with Sparse Attention

Scale to Many VMs & PMs. For effective rescheduling decisions, VMR^{2}L must extract meaningful representations of the state observation, which include features of each individual PM and VM as well as their affiliations. As Fig. 1 implies, the number of VMs can vary drastically even in the same cluster. This implies that the size of the features at each time step is also highly dynamic. To encode these features, one option is to concatenate features from all VMs and PMs into a long vector. However, this approach cannot handle an arbitrary number of VMs as neural networks usually require fixed-sized inputs, and it also requires a model with many parameters that would be difficult to train.

Instead, we propose to share two small embedding networks across all VMs and PMs — one to process each PM's features and another one to process each VM's features (Fig. 6 and 7). As such, the number of weight parameters is *independent* of the number of machines in the system. This is achieved via an attention-based transformer model [13, 61] but tailored for rescheduling. Transformers have demonstrated strong performances in Natural Language Processing [15], Computer Vision [21], as well as combinatorial optimization, such as in vector bin-packing [40, 65]. However, compared to bin-packing, there is a notable difference in VM rescheduling: we must choose from a set of VMs that have already been assigned to PMs.

Tree-level Features. Consider a PM with 2 CPUs left. It contains a VM with 4 CPUs and another VM with 2 CPUs. Suppose a second PM has a fragment size of 8 while hosting a VM with 8 CPUs. To minimize the total 16-core fragments, an ideal approach would be to first remove the two VMs with 2 and 4 CPUs from the first PM, and then reassign the VM with 8 CPUs from the second PM to the first PM. However, if we merely include the source PM's features in each of the VM's features and feed them into the vanilla transformer model, there will not be sufficient information for the two actors to take the above actions. Instead, each VM must be aware of the other VMs that are hosted on the same PM, which is not possible in the vanilla transformer model. In fact, each PM can be viewed as a tree of depth one, where the PM acts as the root node and the VMs it hosts act as the leaf nodes. In order to allow every VM to recognize which other VMs are hosted on the same PM, we propose to include an additional stage of *sparse local-attention* within each PM tree, i.e., we only allow PMs and VMs to attend to each other if and only if they belong to the same tree.

Architecture Overview. We modify the vanilla transformer architecture as follows. The model is composed of several attention blocks, where each block includes three stages as shown in Fig. 8:

1. All PMs and VMs exchange information if they belong to the same tree via sparse local-attention.

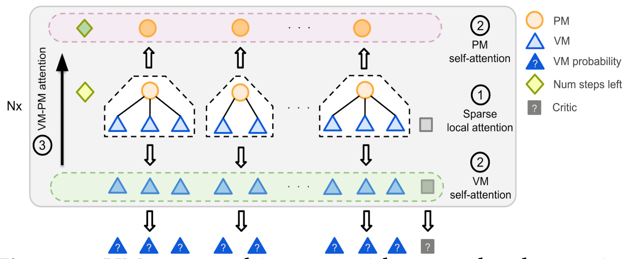


Figure 8. VM actor architecture with sparse local-attention to capture the tree-level features.

2. Each PM attends to other PMs' updated embeddings and each VM attends to other VMs' updated embeddings with self-attention.
3. The new VM embeddings attend to the new PM embeddings through VM-PM attention.

After the three stages, each machine is further processed by two dense layers and layer norm [8]. The updated embeddings are then fed into the next block and the process repeats. Finally, the VM embeddings from the last block are linearly projected into a set of logits followed by Softmax [10] to generate the probability of selecting each VM.

As for the PM actor, we adapt the vanilla encoder-decoder transformer, since we can directly inject the relational information by including the updated VM and PM embeddings from the VM actor as input. We only feed in the selected VM candidate to the encoder, while the decoder still takes in all PMs. Additionally, we also add the VM-PM attention score from stage 3 for the selected VM, since the score indicates which PMs the VM actor attends to and encourages the two actors to better coordinate. The output embeddings of each PM is linearly projected into a logit. Based on the selected VM, we mask out all the illegal PMs by setting their logits to be $-\infty$. The remaining logits are translated into the probability of selecting each PM as the destination PM.

3.4 Risk-seeking Evaluation

VMR is distinct from general RL problems as it allows access to a perfect world model with no uncertainties from the environment. In other words, the simulator can directly compute the final state and the corresponding objective value for a given initial state and action sequence.

To take advantage of this distinction, we propose *risk-seeking evaluation*, which is to **sample multiple trajectories during policy evaluation, and only deploy the one with the highest reward**. Given a trained VMR^{2}L checkpoint, to obtain varied trajectories during inference, we sample actions from the learned policy, $\pi(\cdot|s)$, rather than exclusively selecting the action with the highest probability. Note that different migration trajectories can be generated in parallel if multiple GPUs are available, without significantly affecting the inference time. It is also worth mentioning that the concept of using the best-performing trajectory can be further extended to the training process, known as *risk-seeking training* [51].

Action Thresholding. It is important to note that the learned policy is likely to differ from the optimal policy due to approximation errors. In conventional RL applications, this might not be an issue since only the action assigned with the highest probability is chosen, allowing us to safely ignore the approximation errors. However, in VMR, we must sample actions from $\pi(\cdot|s)$ in order to sample multiple trajectories. Suppose actions with probability less than a threshold ϵ are sub-optimal. Although these actions may not be executed in a single period, they are likely to be performed at some point in the entire trajectory.

Let $p_1 = \min_{s\in S}\sum_{a\in A}\pi_\theta(a|s)\cdot\mathbb{1}\{\pi_\theta(a|s)\le\epsilon\}$ be the lowest total probability of sub-optimal actions over all states. Then, the probability that the agent does not perform any sub-optimal actions along $MNL = 50$ rescheduling steps is upper bounded by $(1-p_1)^{MNL} \le e^{-MNL\cdot p_1}$. If $p_1 = 0.005$, then 23% of the trajectories we sample will contain sub-optimal actions. Therefore, at each migration step, the VM actor computes the probability of selecting each VM candidate, and we calculate a threshold based on the quantile of all VM probabilities. We then mask out all VM candidates that are assigned with probabilities falling below the threshold, and similarly for PMs.

4 Implementation

Datasets. We collect two datasets[9] from real industry-scaled data centers – a Medium dataset with up to 2089 VMs and 280 PMs, and a Large dataset with up to 4546 VMs and 1176 PMs[10]. Each dataset contains 4400 mappings (or instances), which represent the assignments of VMs to PMs at various points in time. To release these datasets while ensuring confidentiality of business operations and avoiding potential train/test leakage, we anonymize each mapping by randomly removing some of the existing VMs and redeploy the VMs to any PMs that they can fit. We split the 4400 mappings into 4000 for training, 200 for validation, and 200 for testing. Our designed Gym simulator directly supports the dataset format. To our knowledge, these are the largest datasets for VM rescheduling based on real traces, and shall be very useful to the community.

Algorithm Specifics. We implement VMR^{2}L based on the CleanRL framework [35] using PPO as the backbone [55].

[9]While a data center might have over 10,000 PMs, they are typically organized into clusters, each consisting of hundreds of PMs, for improved fault isolation and easier management. Migrations usually occur within each cluster. Our Large dataset is already larger than the size of a typical cluster. See Section 1.

[10]Note that the Large dataset has a lower VM to PM ratio, since it has larger average VM sizes. The datasets are collected from real clusters, which are managed by different service teams. This reflects the operational needs of different services.

VMR^{2}L contains about 8.5K lines of Python code. The framework is implemented using PyTorch [24]. The number of model parameters is independent of the number of VMs or PMs, allowing it to scale to large data centers. VMR^{2}L is lightweight – the saved checkpoint has a size less than 2 MB.
Experiment Setup. All models are trained on a Linux server using eight CPU cores (Intel Xeon E5) and one GPU (NVIDIA RTX 3090) [42, 43]. VMR^{2}L takes 92 hours to train, and 1.1 seconds to solve each mapping. We report the average over 3-5 runs with different random seeds and show the confidence intervals in the convergence plots.

5 Evaluation

We conduct extensive experiments to answer:
• How far is VMR^{2}L from the optimal solution? (§ 5.2)
• How much gain does each component provide? (§ 5.3)
• How does the two-stage framework allow VMR^{2}L to accommodate different constraints: i) constraints on the original Medium dataset, ii) muti-dimensional resource constraints, and iii) service affinity constraints? (§ 5.4)
• Can VMR^{2}L optimize i) an objective other than FR, ii) a mixed objective defined with multiple resource types? (§ 5.5)
• How well does VMR^{2}L generalize to i) workloads different from the train distribution, ii) MNLs that are different at inference time, iii) more PMs and VMs, iv) different clusters? Different Workloads with Different MNLs, and (v) varying workloads with different MNLs?(§ 5.6)
• Is A Larger Cluster More Difficult for VMR^{2}L to Learn? (§ 5.7)
• Where do the improvements come from intuitively? (§ 5.8)

5.1 Existing Baseline Algorithms

As later discussed in Section 6, existing baselines can be summarized into six categories: heuristics (e.g., filtering-based heuristic, α-VBPP), optimization algorithms (e.g., MIP), approximate algorithms (e.g., POP), search-based algorithms (e.g., MCTS), deep learning-based (e.g., Decima), and hybrid methods (e.g., NeuPlan). We compare with at least one representative algorithm from each category.
MIP Algorithm: introduced in Section 2.1.
Filtering-Based Heuristic Algorithm (HA): introduced in Section 2.1.
Vector Bin Packing Problem (α-VBPP): We generalize the VBPP [49] algorithm for initial scheduling to rescheduling. We first divide the entire episode into MNL/α stages. During each stage, we greedily remove α number of VMs that lead to the most fragments, and then apply VBPP to treat them as incoming VMs. We carefully tune α (10 in our case) to achieve the best FR reduction.
Partitioned Optimization Problems (POP) [47]: It solves the optimization problem formulated in Section 2.1 by randomly splitting the problem into subproblems (each containing a subset of VMs and PMs), applying an MIP solver to each subproblem, and finally combining the results into a global solution. We choose the smallest number of subproblems (16 in our case) that allows POP to meet the five-second limit.
Monte-Carlo Tree Search (MCTS) [67]: As traditional search based methods need to perform multiple rollouts during inference time to achieve a good performance, we use DDTS [67] to prune the search space.
Decima [44]: Decima uses a graph neural network to encode the VM and PM information and trains using deep RL. Decima balances the size of the action space and the number of actions required by decomposing VM rescheduling decisions into a two-dimensional action, which outputs i) the VM that needs to migrate, and ii) an upper number of PM subsets to choose as the destination.
NeuPlan [66]: In the first stage, an RL agent takes in the problem as a graph and generates the first few VM migrations to prune the search space. In the second stage, it uses an MIP solver for the remaining MNLs. A relax factor β (30 in our case) is used to control the size of the MNL space to be explored by MIP.

5.2 Overall Performance

Fig. 9 shows the FR and inference latency of all methods on the Medium dataset. VMR^{2}L achieves a lower FR compared to all baselines. Notably, VMR^{2}L is merely **2.86%** behind the optimal MIP solution (0.2941 vs. 0.2859) when $MNL = 50$. Meanwhile, VMR^{2}L can generate one trajectory within 1.1s, while MIP requires 50.55 minutes to provide the near-optimal solution. It is worth noting that with higher MNLs, the performance gap between VMR^{2}L and MIP does not increase as significantly as compared to other baselines.

α-VBPP only removes α number of the worst VMs for each stage based on a single timestep, failing to consider future opportunities to replace them back, which leads to its inferior performance. POP fails to achieve good performance since it still relies on MIPs to solve each subproblem. To meet the second-level latency requirement, we must divide the problem into many subproblems, causing its solutions to be only locally optimal. On the other hand, Decima reduces the large action space by limiting the PM actor to only select from a subset of PMs, but the subsampling of PMs is completely random, as opposed to our solution. While MCTS with DDTS uses neural networks to prune the search space, it still requires a significant number of rollouts to achieve stable performance. Lastly, NeuPlan also fails to deliver a satisfying solution for high MNLs, since it can only use MIP to solve a small number of steps in order to meet the latency requirement. Although HA/MCTS achieves competitive results on smaller MNLs (MNL $\leq$ 20), repeating HA/MCTS with smaller MNLs multiple times, as done by α-VBPP, performs poorly on larger MNLs (e.g., MNL=50) because it tends to get stuck in local optima at each individual stage.

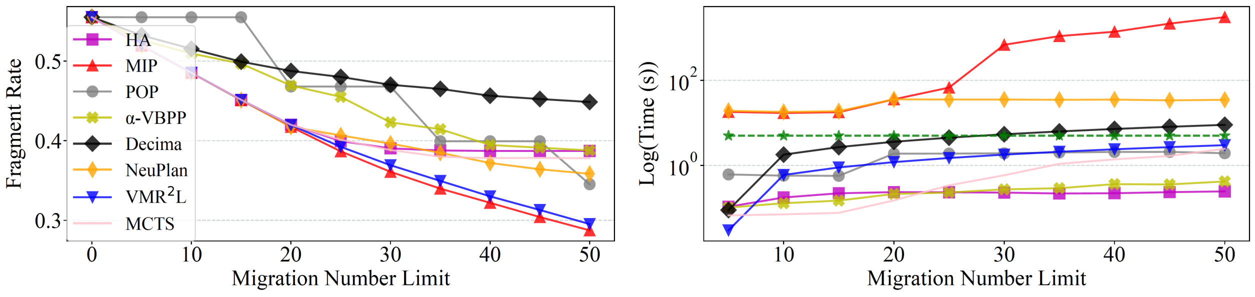
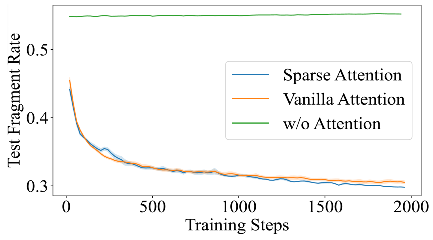
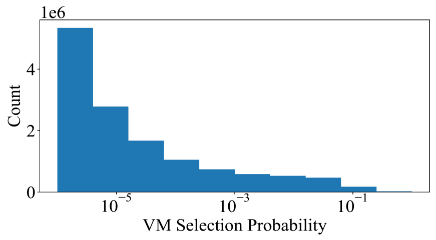


Figure 9. Fragment rate (left) and inference time (right) of VMR^{2}L compared with baselines at different MNLs.





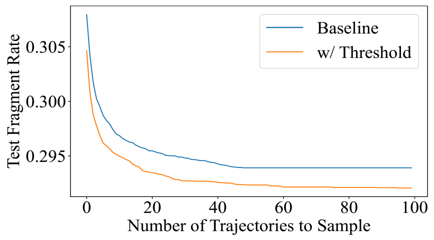


Figure 10. Ablation on Sparse Attention. **Figure 11.** VMs probability distribution. **Figure 12.** Risk-seeking Evaluation.

To summarize, algorithms that involve MIP or searching often fail to deliver a satisfying solution under the strict latency requirement. Heuristic methods are fast but are also suboptimal. Deep learning-based methods can meet the latency requirement since the models do not need to be retrained at inference time, but are difficult to train without the set of customized techniques we proposed for VMR. We provide a case study and a tool to visualize where the improvements of VMR^{2}L come from in Section 5.8.

5.3 Performance Decomposition of VMR^{2}L

We conduct an ablation study using MNL = 50 on the Medium dataset. In summary, FR performance reduces 16.46% without the two-stage framework[11], and improves from 0.3090 and 0.3079 to 0.2941 when sparse attention and risk-seeking are added, respectively. Recall that the near-optimal MIP solution is 0.2859, which means that *Sparse Attention* achieves $\frac{0.3090-0.2941}{0.3090-0.2859} = 64.5\%$, and *Risk-Seeking* achieves $\frac{0.3079-0.2941}{0.3079-0.2859} = 62.7\%$ of the potential room for improvement.

Sparse Attention. We compare against two baselines. *w/o Attention* uses a multilayer perceptron (MLP) [52] as the feature extraction module. MLP concatenates features of all PMs and VMs and thus requires much more trainable parameters that scale linearly with the number of machines in the system. *Vanilla Attention* has fewer parameters as it shares a single small embedding network for all PMs and a second small embedding network for all VMs, but it uses the original encoder-decoder transformer [61, 64] without attending to tree-level features. Fig. 10 shows that MLP fails to converge due to its large number of trainable parameters. As training progresses, *Sparse Attention* learns to capture relational features unique to VMR and gradually outperforms *Vanilla Attention*. We show a case study of how such relational information can benefit VMR intuitively in Section 5.8.

Risk-Seeking Evaluation. At deployment time, given a trained VMR^{2}L checkpoint and an initial vm-pm state, we generate multiple migration plans. We then leverage our simulator to calculate the resulting objective under each plan and only deploy the plan that yields the best outcome. Recall that VMR^{2}L takes 1.1s to generate each trajectory, and suppose we have 8 GPUs to run generations in parallel, generating 16 trajectories would take 2.2s.

To avoid sampling suboptimal actions in the trajectory, we mask out PMs and VMs that are assigned low probabilities. We plot the distribution of probabilities assigned to different VMs in the validation set in Fig. 11. Notice that most VMs are assigned low probabilities. In fact, fewer than 0.8% of VMs have a greater than 1% chance of being selected. Therefore, we compute a quantile $\in \{0.95, 0.98, 0.99, 0.995\}$ for all VMs and all PMs at each step and mask out all machines that have probabilities that fall below the corresponding threshold. We perform a grid search on the two quantiles using the validation set and apply the best combination to the test set. Fig. 12 shows the final FR decreases when we sample more trajectories, and the FR decreases further after we apply thresholding.

5.4 Different Constraints with Two-Stage Framework

More Resource Constraints. To analyze how the two-stage framework supports different constraints, we compare it with two baselines: **i) Penalty**: a penalty of -5 is given if the

[11]The results for the two-stage framework are shown in Section 5.4.

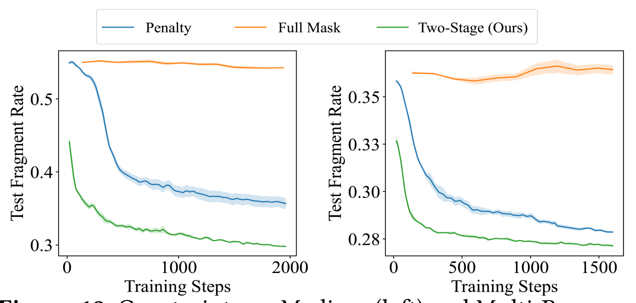


Figure 13. Constraints on Medium (left) and Multi-Resource (right).

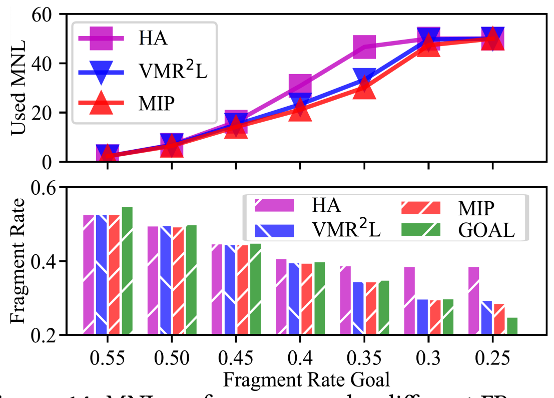


Figure 14. MNL performance under different FR goals.

Table 2. FR under different affinity constraint levels.

Aff. Level	0	1	2	3	4	8
Aff. Ratio	0	1.12%	1.86%	3.46%	6.50%	38.3%
VMR^{2}L	0.3032	0.3029	0.3034	0.3048	0.3045	0.3306
MIP	0.2859	0.2860	0.2860	0.2862	0.2864	OOT

agent takes an illegal action, and **ii) Full-Mask**: the VM candidate and the PM destination are generated simultaneously, with all illegal pairs having probabilities of zero.

In addition to the Medium dataset, we consider traces from an additional data center that is smaller but has more complicated multi-dimensional resource constraints with more VM and PM types. The new *Multi-Resource* dataset has two PM types — one has 88 CPUs and 256 GB of memory, and the other has 128 CPUs and 364 GB of memory. The regular VM types in Table 1 always have a CPU-memory ratio of 1 : 2. For memory-intensive applications, a user might request additional memory, and the CPU-memory ratio can increase up to 1 : 8.

As we can see from Fig. 13, **Penalty** suffers from a slower convergence to a sub-optimal level, since the large negative penalties required to eliminate illegal actions tend to dominate the gradient signal, especially during early stages of training. **Full-Mask** does not converge under both constraint settings, since its action space is the product of the number of VMs and the number of PMs, which leads to poor exploration. In comparison, **Two-Stage** decomposes the action space by designating stage 1 to focus on the set of VM candidates and stage 2 to focus on the set of PM destinations, resulting in much faster convergence.

Service Constraints. We consider a practical constraint in the form of a hard anti-affinity, where a VM cannot be placed on the same PM with some other selected VMs. It can i) prevent resource-intensive VMs to be hosted together, which leads to performance interference, and ii) support critical services that require backup VMs in case of a PM failure. To enforce anti-affinity, we mask out all PMs that host conflicting VMs in stage 2 after selecting a VM candidate in stage 1. We typically observe an affinity ratio requirement to be under 5% in real-world traces. Affinity ratio means the average percentage of VMs that a given VM conflicts with. We synthesize the service anti-affinity constraint on the Medium dataset.

In Table 2, we see that VMR^{2}L maintains consistent performance under typical affinity ratio levels. To demonstrate the robustness of VMR^{2}L to extreme constraint levels, we further evaluate it when the affinity ratio surges to 38.3% and see that VMR^{2}L is still able to achieve a reasonable FR.

5.5 Objectives Generalization

5.5.1 Minimize MNL Given FR Goals.

VMR^{2}L's flexibility enables it to learn different policies depending on the high-level objective. We now consider a new objective: minimize the number of VM migrations to reach a given FR goal, to reduce migration costs. To support this objective, we simply modify the original reward function (Equation 9) as follows:

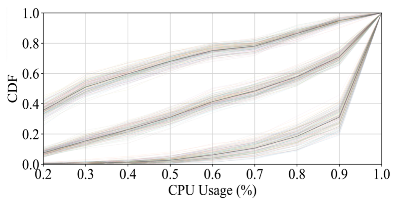
$$R_{fr} = (S_{\text{before, src}} - S_{\text{after, src}}) + (S_{\text{before, dest}} - S_{\text{after, dest}}), \quad (10)$$

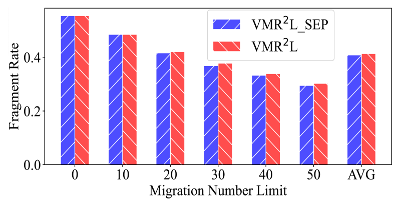
$$R = \begin{cases} -1 + R_{fr}, & FR > FR_{\text{Goal}}, \\ 10 + R_{fr}, & FR \leq FR_{\text{Goal}}. \end{cases} \quad (11)$$

On top of the original reward, we add a penalty of -1 if the FR is above the goal as it indicates additional VM migrations are required, and a bonus of +10 if VMR^{2}L reaches the goal. In Fig. 14, the top subfigure shows the number of migration steps, while the bottom subfigure shows the achieved FR, both sharing the x-axis as different FR goals. On average, MIP and VMR^{2}L achieve 14.77% and 11.11% fewer MNLs than HA, respectively. VMR^{2}L requires only 3.66% more VM migrations, but with second-level solution time.

5.5.2 Mixed Objective (i): Multi-VM-Type FR.

What if a cluster wants to optimize for multiple VM types, such as 16xlarge VMs in addition to the original 4xlarge VMs? Recall that 4xlarge requires 16 cores on a single NUMA, while 16xlarge requires 64 cores deployed across two NUMAs, introducing additional complexities. Let FR_{16} denote 16-core FR, and FR_{64} denote 64-core FR. We optimize for the objectives as the convex combinations of FR_{16} and FR_{64}:





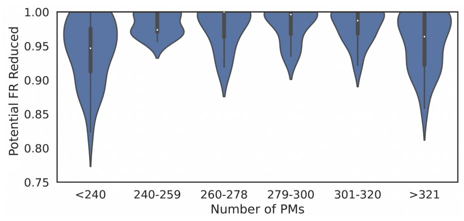


Figure 15. CPU usage on PMs under different workloads.

Figure 16. FR gap between VMR^{2}L and VMR^{2}L $_{\text{SEP}}$.

Figure 17. Ratio of potential FR achieved when deploying on clusters with different numbers of PMs.

Table 3. Mixed objective (i) with FR_{16} and FR_{64}.

	λ	0	0.2	0.4	0.6	0.8	1
VMR^{2}L	FR_{16}	0.2941	0.3079	0.3413	0.4086	0.4214	0.4532
	FR_{64}	0.9478	0.9263	0.6960	0.5900	0.5603	0.5473
	Obj$_\lambda$	0.2941	0.4316	0.4832	0.5174	0.5325	0.5473
POP	FR_{16}	0.3447	0.3650	0.3971	0.3992	0.3991	0.5215
	FR_{64}	0.9836	0.8235	0.7312	0.7280	0.7278	0.7222
	Obj$_\lambda$	0.3447	0.4567	0.5308	0.5964	0.6621	0.7222

Table 4. Mixed objective (ii) with FR_{16} and Mem_{64}.

	λ	0	0.2	0.4	0.6	0.8	1
VMR^{2}L	FR_{16}	0.2709	0.2700	0.2872	0.3071	0.3127	0.3182
	Mem_{64}	0.3032	0.2955	0.2695	0.2480	0.2490	0.2449
	Obj$_\lambda$	0.2709	0.2751	0.2802	0.2716	0.2617	0.2449
POP	FR_{16}	0.2808	0.2809	0.2843	0.3055	0.3073	0.4464
	Mem_{64}	0.3107	0.2832	0.2762	0.2559	0.2550	0.2537
	Obj$_\lambda$	0.2808	0.2813	0.2811	0.2757	0.2655	0.2537

$$\text{Obj}_\lambda = \lambda \cdot FR_{64} + (1-\lambda) \cdot FR_{16}, \tag{12}$$

where λ is a predefined parameter based on which VM types to prioritize. Table 3 presents the results when we optimize for $\lambda \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$ on the *Multi-Resource* dataset introduced in Section 5.4. For both methods, as λ increases, some FR_{16} has to be sacrificed when the objective emphasizes more on FR_{64}. Overall, in terms of Obj$_\lambda$, we see that VMR^{2}L consistently outperforms POP under different λ's. Note that when $\lambda = 0.2$, Obj$_\lambda$ is lower for VMR^{2}L, but POP achieves lower FR_{64}. This is because to minimize a mixed objective, different algorithms may choose to focus on each individual objective differently, even under the same λ. Therefore, individual objectives might not be directly comparable.

5.5.3 Mixed Objective (ii): Multi-Resource-Type FR.

In some clusters, the optimization target can be defined in terms of multiple resource types. We show that VMR^{2}L is capable of handling such a multi-dimensional objective by using 16-core CPU fragments and 64-GB Memory fragments (denoted as Mem_{64}) as an example. Mem_{64} refers to a discrete resource fragment representing 64 GB of memory capacity. The objective function can be written in a form similar to Equation 12. We show the results on the *Multi-Resource* dataset in Table 4. VMR^{2}L still consistently outperforms POP in terms of the mixed objective. Another interesting observation is when we increase λ from 0 to 0.2, both FR_{16} and Mem_{64} decreased for VMR^{2}L. We hypothesize that rewarding the model for additional objectives helps to make the reward less sparse, thereby further stabilizing RL training [54].

Table 5. Generalization to abnormal workloads. POP is a baseline algorithm for large-scale resource allocation [47].

Methods	L (MNL=100)	M (MNL=100)	H (MNL=50)
HA	0.256(-2.7%)	0.276(-8.0%)	0.387(-10.9%)
VMR^{2}L (L)	**0.237(+4.8%)**	0.261(-2.7%)	0.424(-18.6%)
VMR^{2}L (M)	0.239(+4.0%)	0.238(+6.3%)	0.422(-18.2%)
VMR^{2}L (H)	0.243(+2.4%)	0.248(+2.4%)	**0.303(+12.2%)**
VMR^{2}L (L,H)	**0.237(+4.8%)**	**0.237(+6.7%)**	0.326(+5.5%)
POP	0.249	0.254	0.345

5.6 Generalization and Scalability of VMR^{2}L

5.6.1 Abnormal Workloads.

It is well-known that common DRL applications often experience performance degradation due to distribution shifts [39]. However, in real-world service traces, there are periods, such as during deadlines or holidays, where workloads (defined as the percentage of available CPUs on PMs) deviate significantly from the norm. To assess whether VMR^{2}L can adapt to these abnormal workload levels, we gathered two additional datasets representing *Low*(L) and *Middle*(M) workloads. In our study, the medium dataset represents *High*(H) workloads. The workload distributions of each train mapping from the three datasets are shown in Fig. 15. Note that these three datasets have strictly non-overlapping workload distributions, i.e., we cannot find a training sample from *High* that has a workload similar to *Middle* or *Low*.

We train VMR^{2}L on one or mixed workload datasets and evaluate the FR on each individual workload level. The results are summarized in Table 5. For example, (L,H) means that we train VMR^{2}L on both *Low* and *High* datasets. We set $MNL = 100$ for L and M since the FR difference is low until MNL increases to 100. As MIP runs out of time due to the larger MNL used on L and M, we choose POP as the

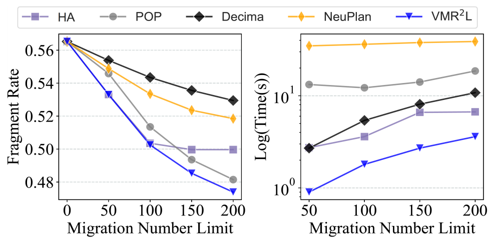
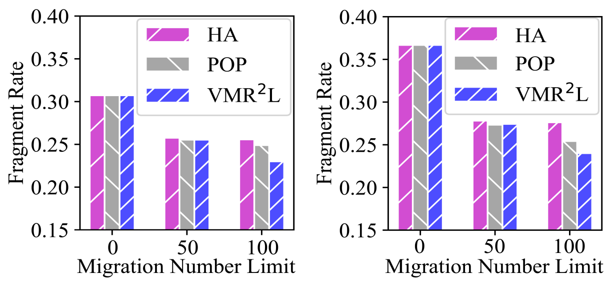


Figure 18. FR and time performance on the Large dataset.



(a) FR on the low workload. (b) FR on the medium workload.

Figure 19. FR on different workloads.

standard baseline since it is easy to tune and exhibits strong FR performances. First, we see that VMR^{2}L outperforms the two baselines when trained on the same workload level as test. When VMR^{2}L trains with workloads smaller than the test workload, VMR^{2}L suffers from performance degradation. Intuitively, a high workload requires the agent to create available resources by first removing existing VMs away from the destination PM, but this action is less common on lower workloads and thus cannot be effectively learned.

Remarkably, when trained on both L and H, VMR^{2}L can learn a general policy and perform the best on M *without ever experiencing middle workloads*. Thus, we recommend end-users to train VMR^{2}L with a combination of high and low workloads from their data centers. This allows VMR^{2}L to bridge gaps in the training data and better generalize when encountering abnormal workloads.

5.6.2 Generalizing to Different MNLs. In practical scenarios, MNL often fluctuates due to varying business needs, such as when in the day VMR is performed. We show that training a single VMR^{2}L agent with $MNL = 50$ can yield effective results across a range of MNLs $\in \{10, 20, 30, 40, 50\}$. To compare, we train a separate VMR^{2}L agent for each MNL, denoted as VMR^{2}L $_{\text{SEP}}$. As shown in Fig. 16, VMR^{2}L performs only marginally worse than VMR^{2}L $_{\text{SEP}}$ with an average FR performance gap of 1.16%. This suggests that the VMR^{2}L agent trained with a large MNL can be readily applied to tasks with smaller MNLs. It avoids the overhead of maintaining a separate VMR^{2}L agent for each MNL.

5.6.3 Generalizing to Different Clusters. We evaluate the generalization ability of the VMR^{2}L when trained on a specific cluster and deployed to different clusters. Specifically, we use the VMR^{2}L model trained on the Medium dataset, which contains 280 PMs. To simulate varying cluster sizes, we randomly modify the Medium dataset by adding or removing PMs, generating a total of 100 different mappings.

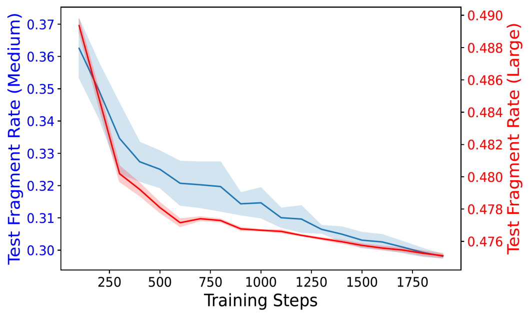
Fig. 17 illustrates the potential FR achieved by VMR^{2}L on clusters with different numbers of PMs. The potential FR is defined as the difference between the initial FR and the FR achieved by MIP. Our results show that VMR^{2}L maintains nearly the same performance when deployed on clusters with a PM count that varies by less than 10%. Performance remains robust even with variations of 10% to 20%, achieving over 95% of the potential FR. However, when the number of PMs differs by more than 20%, a slight decline in performance is observed. Even in this scenario, VMR^{2}L still significantly outperforms POP, which achieves only around 78% and has to be retrained on each cluster.

5.6.4 More VMs & PMs. To see how VMR^{2}L scales to a larger cluster, we conduct experiments on the Large dataset with 4546 VMs and 1176 PMs. Fig. 18 shows the performance against different baselines when MNL varies from 50 to 200. MIP is not included here since it takes more than an hour to solve a single migration path. Similar to the Medium dataset, VMR^{2}L achieves lower FRs than the baselines, with an inference time of 3.8s to solve one mapping.

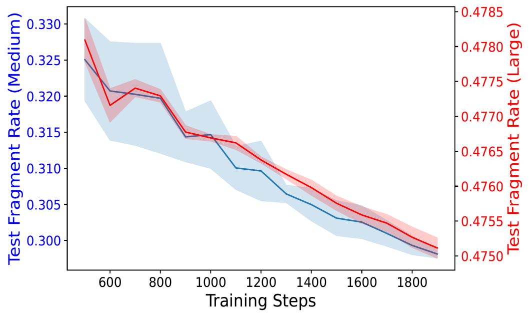
5.6.5 Different Workloads with Different MNLs. We evaluate VMR^{2}L with varying workloads under different MNLs. We set MNL=100 when evaluating the Low and Middle workload datasets since the FR difference is low until we increase MNL to 100. From Fig. 19, we can see HA, POP, and VMR^{2}L can all decrease FR at MNL=50. However, HA fails to decrease FR at MNL=100. Instead, VMR^{2}L achieves 7.42% and 4.8% lower FR on the low workload, and 13.77% and 6.3% lower FR on the middle workload compared to HA and POP, respectively. These results demonstrate that VMR^{2}L is capable of handling varying workloads.

5.7 Is A Larger Cluster More Difficult for VMR^{2}L to Learn?

Recall that the Medium dataset has up to 2089 VMs and 280 PMs, and the Large dataset has up to 4546 VMs and 1176 PMs. We demonstrate that larger clusters are not inherently more difficult to train in VMR^{2}L by comparing the convergence speeds on the Medium and Large datasets. The results are shown in Fig. 20(a). Since the initial and optimal FR values are different between the two datasets, we use a dual y-axis to plot the convergence curves. At first glance, it may seem that convergence is slower on the Medium dataset (blue) compared to the Large dataset (red), which might seem counterintuitive given the higher number of VMs and PMs in the latter. However, we hypothesize that this is because there are smaller VMs in the Large dataset. Since smaller



(a) Including the initial stage.



(b) After the initial stage.

Figure 20. Convergence speed on different cluster sizes.

VMs are easier to move around, it is easier to learn a simple policy that can effectively reduce these fragments in the initial stages. To test this, we replot the convergence curves in Fig. 20(b), excluding the initial stage where most of the "low-hanging fruits" have been picked. After the initial stage, VMR^{2}L converges slightly faster on the medium dataset, but the difference is very minimal and should not pose an issue. Note that since we use a dual y-axis, the absolute slopes are no longer comparable, but instead we shall focus on the similar linear downward trend in both datasets.

5.8 Intuitions Behind VMR^{2}L: A Case Study

Intuitively, where do the improvements of VMR^{2}L come from? To answer this question, we build a tool to visualize which VM is being migrated at each step. We randomly select one mapping from 200 test mappings on the Medium dataset and analyze how VMR^{2}L reduces FR when $MNL = 50$. VMR^{2}L optimizes FR from a global perspective. In Fig. 21, we analyze the three PMs involved during steps 38-40. Each PM consists of two NUMA nodes, represented by the two vertically stacked bars. Different colors indicate the total allocated size of each VM type on a NUMA. For example, in the first NUMA at step 38, the orange section has a total size of 28 cores. Since an xlarge VM requires 4 cores, this implies that there are seven xlarge VMs occupying that space. The gray regions represent unused resources (free space).

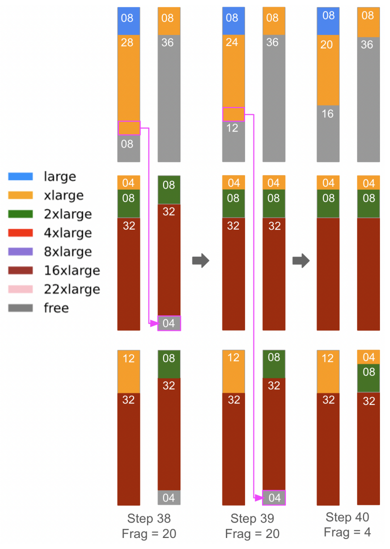


Figure 21. VM-PM Migration Details. Each equal-sized rectangle represents a NUMA node within a PM. Different colors indicate the total allocated size of a VM type on each NUMA.

At step 38, VMR^{2}L removes a 4-core VM (from the orange section) from the top PM, which eliminates fragmentation on the destination NUMA but temporarily creates four fragments on the source NUMA. This results in a net reward of zero at this step. At step 39, the agent identifies a second 4-core VM on the source NUMA and migrates it to another PM, effectively eliminating all remaining fragments on both source and destination NUMAs. The reason the color distribution appears to change after migration is that the total allocated size of VMs on a NUMA is updated after each migration. The individual VMs remain the same, but their placement alters the total distribution of resources on each NUMA. This process highlights how sparse attention enables the agent to recognize multiple rescheduling opportunities and make globally optimal decisions for reducing FR across the entire system. This example shows that VMR^{2}L is able to sacrifice immediate rewards for long-term FR performance due to the cumulative reward design in RL.

6 Related Work

Connections to Bin Packing. The use of optimized placement mechanisms proved to be successful in a broad set of use cases, including production quality scenarios [6] as well as transportation logistics [12, 22, 34, 63]. A typical solution exploits heuristics based on bin packing [49]. In fact, VM placement can be modeled as a bin-packing problem, where VMs and PMs are objects and bins, respectively. Bin packing typically involves packing a set of items into fixed-sized bins such that the number of bins required [12] or the total surface area is minimized [22, 34]. However, there are two notable differences. First, the problem of VM rescheduling concerns adjusting an initial assignment of VMs to PMs. On the other hand, rebalancing items already packed in bins has received little attention in the context of other bin-packing applications. A critical aspect of the initial assignment is the current VM affiliations, which existing bin-packing solutions often do not consider but we show is critical to VMR via *tree-level features*. Second, the total number of VMs and PMs in a data center can easily go into the range of several thousand or more [63] and is far more than the typical scale of bin packing problems, which typically involve no more than a few hundred items [41, 67].

RL for Optimization Problems. RL has been recently introduced to solve optimization problems, e.g., building ML compilers and optimizing neural network architectures [29]. In particular, RL is used to select branching variables or find cutting planes in the Branch-and-cut method [23, 25, 26]. Besides, RL can also be applied to existing heuristics for MIPs to further increase the quality of solutions [9, 58]. In fact, most state-of-the-art solutions for optimization problems often involve MIP or searching [66], but these methods are not directly appropriate for the VM rescheduling task due to their poor computation complexity. Although they are designed to accelerate MIPs, as shown in Section 5.2 even a state-of-the-art technique such as POP [47] fails to deliver a satisfying solution within the second-level time limits of the VM rescheduling task. While learning-based methods [44] can meet the latency requirement by leveraging their generalization ability at deployment to avoid retraining, they do not involve techniques that are tailored for VMR, which we propose in VMR^{2}L.

7 Discussion

Noisy Neighbors. A challenge in rescheduling is performance interference caused by noisy neighbors, which are VMs that disproportionately consume shared resources, leading to degradation for other VMs on the same PM. Our approach can address this issue by incorporating multi-resource constraints or hard anti-affinity policies (Section 5.4), ensuring that certain VMs are not hosted on the same PM to avoid resource contention. However, these strategies require prior knowledge of the resource profiles of the VMs involved and the ability to isolate resources accordingly. Future work may consider resource reservation to ensure dedicated resources without interference. Additionally, developing predictive models for workload characterization can help anticipate resource demands and interference patterns of different VMs, enabling more dynamic workload management.

Adapting to New data. Section 5.6 shows VMR^{2}L has strong generalization ability, learning a policy that performs well across different workload levels not present in the training data without retraining or even finetuning. It also generalizes to different MNLs by training a single agent with a larger MNL and applying it across smaller MNLs, avoiding the need for separate agents (Section 5.6.2). If we see large distribution shifts or performance drops, VMR^{2}L readily supports off-the-shelf finetuning methods (e.g., top-layer finetuning [32], adding adapters [31], LoRA [33]).

Efficient Training in Deterministic Environments. VM rescheduling differs from typical RL tasks [18, 19] that require large datasets due to stochastic transitions. In contrast, VM rescheduling involves deterministic transitions, where the outcome is fully predictable given a specific state and action. We only require the initial VM mappings for training, making it more data-efficient. In light of this, our work introduces a simulator that enables offline training, fully simulating the rescheduling environment and allowing agents to learn policies without requiring extensive real-world data.

Broader Insights. While immediate application is scheduling, many system problems share the same underlying principles (e.g., large-scale, no environmental uncertainties, and strict latency requirements). Our RL formulation, and techniques such as action decomposition and risk-seeking evaluation, are transferable and could inspire other system applications facing similar challenges.

8 Conclusion

Compared to conventional bin-packing applications, VM rescheduling presents unique challenges due to the expanding size of data centers. It must handle a large volume of VM requests while meeting a second-level inference speed, given the dynamic nature of VM states. As such, we propose VMR^{2}L, a deep RL approach designed specifically for VM rescheduling: i) a two-stage framework to seamlessly accommodate different service constraints, ii) a sparse attention module to better capture local VM-PM relations, and iii) risk-seeking evaluation to offer a better trade-off between speed and performance. Future work includes optimizing for best-case performance during training [51], which could better align with our risk-seeking evaluation pipeline. Additionally, incorporating the estimated remaining runtime of each VM and future VM demands [44] could further enhance performance. Predicting resource usage patterns may also help prevent performance inference by VMs hosted on the same PM. Furthermore, our current action design requires the agent to migrate VMs one at a time. Permitting the agent to swap multiple VMs simultaneously could simplify the identification of a feasible migration path. Overall, we hope our released datasets and RL environment will facilitate future research in this direction.

Acknowledgement

We would like to thank our shepherd Mangpo Phothilimthana and anonymous EuroSys reviewers for their valuable comments and insightful feedback.

References

[1] Cluster design in data center. https://core.vmware.com/resource/vsan-cluster-design-large-clusters-versus-small-clusters#section1.

[2] Cplex optimizer. https://www.ibm.com/analytics/cplex-optimizer.

[3] Gurobi solver. https://www.gurobi.com/.

[4] Kubernetes scheduler. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.

[5] Numa architecture platforms. https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/17_NUMA_Architecture_Platforms/NUMA_Architecture_Platforms.html. Accessed: 2024-10-05.

[6] Ahmad, R. W., Gani, A., Hamid, S. H. A., Shiraz, M., Yousafzai, A., and Xia, F. A survey on virtual machine migration and server consolidation frameworks for cloud data centers. *Journal of network and computer applications 52* (2015), 11–25.

[7] An, Z., Ding, X., and Du, W. Go beyond black-box policies: Rethinking the design of learning agent for interpretable and verifiable hvac control. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (2024), pp. 1–6.

[8] Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization, 2016.

[9] Barrett, T., Clements, W., Foerster, J., and Lvovsky, A. Exploratory combinatorial optimization with reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2020), vol. 34, pp. 3243–3250.

[10] Bishop, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*, 1 ed. Springer, 2007.

[11] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym, 2016.

[12] Cai, Q., Hang, W., Mirhoseini, A., Tucker, G., Wang, J., and Wei, W. Reinforcement learning driven heuristic optimization. *Workshop on Deep Reinforcement Learning for Knowledge Discovery (DRL4KDD) abs/1906.06639* (2019).

[13] Chen, Y., Yang, K., An, Z., Holder, B., Paloutzian, L., Bali, K. M., and Du, W. Marlp: Time-series forecasting control for agricultural managed aquifer recharge. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (2024).

[14] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2* (Berkeley, CA, USA, 2005), USENIX Association, pp. 273–286.

[15] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805v2* (2018).

[16] Ding, X., An, Z., Rathee, A., and Du, W. A safe and data-efficient model-based reinforcement learning system for hvac control. *IEEE Internet of Things Journal* (2025).

[17] Ding, X., Cerpa, A., and Du, W. Exploring deep reinforcement learning for holistic smart building control. *ACM Transactions on Sensor Networks 20*, 3 (2024), 1–28.

[18] Ding, X., Cerpa, A., and Du, W. Multi-zone hvac control with model-based deep reinforcement learning. *IEEE Transactions on Automation Science and Engineering* (2024).

[19] Ding, X., and Du, W. Optimizing irrigation efficiency using deep reinforcement learning in the field. *ACM Transactions on Sensor Networks 20*, 4 (2024), 1–34.

[20] Ding, X., Zhang, Y., Chen, B., Ying, D., Zhang, T., Chen, J., Zhang, L., Cerpa, A., and Du, W. Vmr2l: Virtual machines rescheduling using reinforcement learning in data centers, 2023.

[21] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations* (2021).

[22] Duan, L., Hu, H., Qian, Y., Gong, Y., Zhang, X., Wei, J., and Xu, Y. A multi-task selected learning approach for solving 3d flexible bin packing problem. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems* (Richland, SC, 2019), AAMAS '19, International Foundation for Autonomous Agents and Multiagent Systems, p. 1386–1394.

[23] Etheve, M., Alès, Z., Bissuel, C., Juan, O., and Kedad-Sidhoum, S. Reinforcement learning for variable selection in a branch and bound algorithm. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research* (2020), Springer, pp. 176–185.

[24] Facebook AI Research. Pytorch: An open source machine learning framework. https://pytorch.org/, 2019. Accessed: April 23, 2023.

[25] Gasse, M., Chételat, D., Ferroni, N., Charlin, L., and Lodi, A. Exact combinatorial optimization with graph convolutional neural networks. *Advances in Neural Information Processing Systems 32* (2019).

[26] Gupta, P., Gasse, M., Khalil, E., Mudigonda, P., Lodi, A., and Bengio, Y. Hybrid models for learning to branch. *Advances in neural information processing systems 33* (2020), 18087–18097.

[27] Ha, C. T., Nguyen, T. T., Bui, L. T., and Wang, R. An online packing heuristic for the three-dimensional container loading problem in dynamic environments and the physical internet. In *Applications of Evolutionary Computation: 20th European Conference, EvoApplications 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, Part II 20* (2017), Springer, pp. 140–155.

[28] Hadary, O., Marshall, L., Menache, I., Pan, A., Greeff, E. E., Dion, D., Dorminey, S., Joshi, S., Chen, Y., Russinovich, M., et al. Protean: Vm allocation service at scale. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (2020).

[29] Haj-Ali, A., Huang, Q. J., Xiang, J., Moses, W., Asanovic, K., Wawrzynek, J., and Stoica, I. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems 2* (2020), 70–81.

[30] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence* (2018), vol. 32.

[31] Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., Attariyan, M., and Gelly, S. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning* (2019).

[32] Howard, J., and Ruder, S. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Melbourne, Australia, July 2018), I. Gurevych and Y. Miyao, Eds., Association for Computational Linguistics, pp. 328–339.

[33] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[34] Hu, H., Zhang, X., Yan, X., Wang, L., and Xu, Y. Solving a new 3d bin packing problem with deep reinforcement learning method, 2017.

[35] Huang, S., Dossa, R. F. J., Ye, C., Braga, J., Chakraborty, D., Mehta, K., and Araújo, J. G. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research 23*, 274 (2022), 1–18.

[36] Janani, K., Anuhya, K., Manaswini, V. L., Likitha, V., Suneetha, B., and Vignesh, T. Analysis of ci/cd application in kubernetes architecture. *Mathematical Statistician and Engineering Applications 71*, 4 (Mar. 2023), 11091–11097.

[37] Kaelbling, L. P., Littman, M. L., and Moore, A. W. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research 4* (1996).

[38] Kumar, D., and Li, S. Separating storage and compute with the databricks lakehouse platform. In *2022 IEEE 9th International Conference on Data Science and Advanced Analytics (DSAA)* (2022), pp. 1–2.

[39] Levine, S., Kumar, A., Tucker, G., and Fu, J. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *ArXiv abs/2005.01643* (2020).

[40] Li, D., Ren, C., Gu, Z., Wang, Y., and Lau, F. Solving packing problems by conditional query learning, 2020.

[41] Li, X., Yuan, M., Chen, D., Yao, J., and Zeng, J. A data-driven three-layer algorithm for split delivery vehicle routing problem with 3d container loading constraint. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining* (New York, NY, USA, 2018), KDD '18, Association for Computing Machinery, p. 528–536.

[42] Lin, M., and Jeon, H. Understanding oversubscribed memory management for deep learning training. In *Proceedings of the 5th Workshop on Machine Learning and Systems* (New York, NY, USA, 2025), EuroMLSys '25, Association for Computing Machinery.

[43] Lin, M., Zhou, K., and Su, P. Drgpum: Guiding memory optimization for gpu-accelerated applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (New York, NY, USA, 2023), Association for Computing Machinery.

[44] Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., and Alizadeh, M. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication.* 2019, pp. 270–288.

[45] Mazyavkina, N., Sviridov, S., Ivanov, S., and Burnaev, E. Reinforcement learning for combinatorial optimization: A survey, 2020.

[46] Mergen, M. F., Uhlig, V., Krieger, O., and Xenidis, J. Virtualization for high-performance computing. *SIGOPS Oper. Syst. Rev. 40*, 2 (apr 2006), 8–11.

[47] Narayanan, D., Kazhamiaka, F., Abuzaid, F., Kraft, P., Agrawal, A., Kandula, S., Boyd, S., and Zaharia, M. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021).

[48] Padberg, M., and Rinaldi, G. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review 33*, 1 (1991), 60–100.

[49] Panigrahy, R., Talwar, K., Uyeda, L., and Wieder, U. Heuristics for vector bin packing. *research. microsoft. com* (2011).

[50] Parreño, F., Alvarez-Valdés, R., Tamarit, J. M., and Oliveira, J. F. A maximal-space algorithm for the container loading problem. *INFORMS Journal on Computing 20*, 3 (2008), 412–422.

[51] Petersen, B. K., Larma, M. L., Mundhenk, T. N., Santiago, C. P., Kim, S. K., and Kim, J. T. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *International Conference on Learning Representations* (2021).

[52] Popescu, M.-C., Balas, V. E., Perescu-Popescu, L., and Mastorakis, N. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems 8*, 7 (2009), 579–588.

[53] Rengarajan, D., Vaidya, G., Sarvesh, A., Kalathil, D., and Shakkottai, S. Reinforcement learning with sparse rewards using guidance from offline demonstration. *arXiv preprint arXiv:2202.04628* (2022).

[54] Riedmiller, M., Hafner, R., Lampe, T., Neunert, M., Degrave, J., van de Wiele, T., Mnih, V., Heess, N., and Springenberg, J. T. Learning by playing solving sparse reward tasks from scratch. In *Proceedings of the 35th International Conference on Machine Learning* (10–15 Jul 2018), J. Dy and A. Krause, Eds., vol. 80 of *Proceedings of Machine Learning Research*, PMLR, pp. 4344–4353.

[55] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[56] Shirvani, M. H., Rahmani, A. M., and Sahafi, A. A survey study on virtual machine migration and server consolidation techniques in dvfs-enabled cloud datacenter: taxonomy and challenges. *Journal of King Saud University-Computer and Information Sciences 32*, 3 (2020).

[57] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of go without human knowledge. *nature* (2017).

[58] Song, J., Yue, Y., Dilkina, B., et al. A general large neighborhood search framework for solving integer linear programs. *Advances in Neural Information Processing Systems 33* (2020), 20012–20023.

[59] Talebian, H., Gani, A., Sookhak, M., Abdelatif, A. A., Yousafzai, A., Vasilakos, A. V., and Yu, F. R. Optimizing virtual machine placement in iaas data centers: taxonomy, review and open issues. *Cluster Computing 23* (2020), 837–878.

[60] Thalheim, J., Okelmann, P., Unnibhavi, H., Gouicem, R., and Bhatotia, P. Vmsh: hypervisor-agnostic guest overlays for vms. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), pp. 678–696.

[61] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems 30* (2017).

[62] Wood, T., Ramakrishnan, K. K., Shenoy, P., Van Der Merwe, J., Hwang, J., Liu, G., and Chaufournier, L. Cloudnet: dynamic pooling of cloud resources by live wan migration of virtual machines. *IEEE/ACM Trans. Netw. 23*, 5 (Oct. 2015), 1568–1583.

[63] Xia, Y., Tsugawa, M., Fortes, J. A., and Chen, S. Large-scale vm placement with disk anti-colocation constraints using hierarchical decomposition and mixed integer programming. *IEEE Transactions on Parallel and Distributed Systems 28*, 5 (2016), 1361–1374.

[64] Yang, K., Chen, Y., and Du, W. OrchLoc: In-Orchard Localization via a Single LoRa Gateway and Generative Diffusion Model-based Fingerprinting. In *ACM MobiSys* (2024).

[65] Zhang, J., Zi, B., and Ge, X. Attend2pack: Bin packing through deep reinforcement learning with attention. *ArXiv abs/2107.04333* (2021).

[66] Zhu, H., Gupta, V., Ahuja, S. S., Tian, Y., Zhang, Y., and Jin, X. Network planning with deep reinforcement learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), pp. 258–271.

[67] Zhu, Q., Li, X., Zhang, Z., Luo, Z., Tong, X., Yuan, M., and Zeng, J. Learning to pack: A data-driven tree search algorithm for large-scale 3d bin packing problem. In *Proceedings of the 30th ACM International Conference on Information Knowledge Management* (New York, NY, USA, 2021), CIKM '21, Association for Computing Machinery, p. 4393–4402.