This article has been accepted for publication in IEEE Transactions on Services 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/TSC.2024.3442544

Tetris: Proactive Container Scheduling for Long-Term Load Balancing in Shared Clusters

Fei Xu, *Member, IEEE*, Xiyue Shen, Shuohao Lin, Li Chen, *Member, IEEE*, Zhi Zhou, *Member, IEEE*, Fen Xiao, Fangming Liu, *Senior Member, IEEE*

Abstract—Long-running containerized workloads (*e.g.,* machine learning), which typically show *time-varying* patterns, are increasingly prevailing in shared production clusters. To improve workload performance, current schedulers mainly focus on optimizing *short-term* benefits of cluster load balancing or *initial container placement* on servers. However, this would inevitably bring many *invalid migrations* (*i.e.*, containers are migrated back and forth among servers over a short time window), leading to significant service level objective (SLO) violations. This paper introduces *Tetris*, a *model predictive control* (MPC)-based container scheduling strategy to proactively migrate long-running workloads for cluster load balancing. Specifically, we first build a discrete-time dynamic model for *long-term* optimization of container scheduling. To solve such an optimization problem, *Tetris* then employs two main components: (1) a container resource predictor, which leverages time-series analysis approaches to accurately predict the container resource consumption; (2) an MPC-based container scheduler that jointly optimizes the cluster load balancing and container migration cost *over a certain sliding time window*. We implement and open source a prototype of *Tetris* based on K8s. Extensive prototype experiments and trace-driven simulations demonstrate that *Tetris* can improve the cluster load balancing degree by up to 77.8% without incurring any SLO violations, compared to the state-of-the-art container scheduling strategies.

Index Terms—Long-running containerized workloads, load balancing, migration cost, container scheduling

1 Introduction

Large production clusters are commonly hosting various *long-running* containerized workloads, ranging from online Web services to machine learning applications [1]. Unlike traditional batched jobs that are typically executed within seconds to minutes, these long-running workloads generally last for hours to months [2]. They are often *stateful* and have stringent service level objectives (SLOs) [3]. However, the *time-varying* request loads of such long-running workloads [4] can cause severe contention of shared cluster resources among containers, leading to cluster load imbalance and thus many potential SLO violations [5]. Therefore, the mainstream cluster schedulers such as Borg [6] and Kubernetes (K8s) [7] enable container scheduling to achieve load balancing [8] in shared production clusters.

Though the existing container scheduling policies such as Sandpiper [8] and Medea [2] perform well in achieving cluster load balancing, there still exist a noticeable number of *invalid migrations* (*i.e.*, containers are migrated back and forth *among* servers over a short time window) in shared production clusters. As evidenced by our motivational analysis (in Sec. 2.2) on both Alibaba cluster traces v2018 and v2022 [9], more than *half* of container migrations are invalid migrations as the time window size is set as 3 hours. Such invalid migrations are likely to make the workloads hosted on the migrated containers suffer from serious SLO violations, leading to unexpected performance interference to the containers that are *co-located* on servers (*i.e.*, migration cost). Meanwhile, our motivation experiments in Sec. 2.2 reveal that invalid migrations can significantly reduce the number of requests processed by an Apache Tomcat Web server hosted on a migrated container by up to 99.8%. Accordingly, it is essential for the cluster scheduler to circumvent invalid migrations, especially for long-running workloads.

Unfortunately, many research efforts have been devoted to making *short-term* scheduling decisions based on the *current* cluster status for workload consolidation [10] or load balancing [11]. Though such scheduling policies can acquire short-term (*e.g.*, the current or upcoming timeslot) benefits, they are oblivious to the *time-varying* resource consumption of long-running workloads, which is actually the *root cause of invalid migrations* as discussed in Sec. 2.2. There have also been recent works on the *long-term* optimization of container scheduling, which are reinforcement learning (RL)-based [5], [12] or control-based [13], [14] scheduling policies, to *partially* tackle the issue of invalid migrations. Nevertheless, these techniques solely focus on optimizing the *initial placement* or resource auto-scaling of containerized workloads (*i.e.*, *where to schedule*) over the entire time period (*i.e.*, *infinite future*). The containers to be scheduled (*i.e.*, *which container to schedule*) and the *migration cost* of containers

- *Fei Xu, Xiyue Shen, Shuohao Lin are with the Shanghai Key Laboratory of Multidimensional Information Processing, School of Computer Science and Technology, East China Normal University, 3663 N. Zhongshan Road, Shanghai 200062, China. Email:* fxu@cs.ecnu.edu.cn.
- *Li Chen is with the School of Computing and Informatics, University of Louisiana at Lafayette, 301 East Lewis Street, Lafayette, LA 70504, USA. E-mail:* li.chen@louisiana.edu.
- *Zhi Zhou is with the Guangdong Key Laboratory of Big Data Analysis and Processing, School of Computer Science and Engineering, Sun Yat-sen University, 132 E. Waihuan Road, Guangzhou 510006, China. E-mail:* zhouzhi9@mail.sysu.edu.cn.
- *Fen Xiao is with Tencent Incorporation, 33 Haitian Second Road, Nanshan District, Shenzhen 518054, China. E-mail:* cherryxiao@tencent.com.
- *Fangming Liu is with Peng Cheng Laboratory, and Huazhong University of Science and Technology, China. E-mail:* fangminghk@gmail.com.

Manuscript received September XX, 2023; revised July XX, 2024.

This article has been accepted for publication in IEEE Transactions on Services 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/TSC.2024.3442544

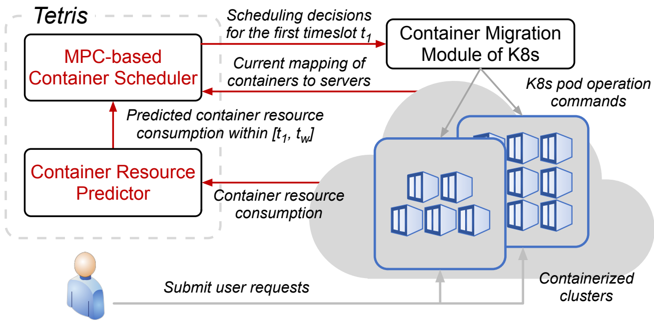


Fig. 1: Overview of *Tetris*. It comprises two pieces of modules including a *Container Resource Predictor* and an *MPC-based Container Scheduler*. By *jointly* optimizing the cluster load balancing degree and container migration cost, *Tetris* decides the appropriate container scheduling plans over a time window $[t_1, t_W]$, which are performed by the *Container Migration Module of K8s* in the containerized cluster.

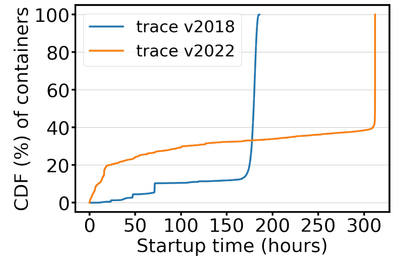


Fig. 2: Container uptime in Alibaba cluster traces v2018 and v2022 [9]. More than 80% of containers last for over 175 hours in the v2018 trace, while around 60% of containers run over 312 hours in the v2022 trace.

have surprisingly received little attention. Such *long-term* optimization techniques can still cause unexpected SLO violations, as evidenced in Sec. 2.3. As a result, scant research attention has been paid to developing container scheduling policies to *fully* deal with the invalid migrations of long-running workloads in production containerized clusters.

To fill this gap, we propose *Tetris* shown in Fig. 1, a model predictive control [15] (MPC)-based container scheduling strategy to proactively make scheduling decisions for long-running containerized workloads. *Tetris* simply optimizes container scheduling *over a certain sliding time window* rather than *the infinite future* (as in RL-based method [3]) for two reasons: *First*, the prediction accuracy of container resource consumption dramatically decreases as the prediction window size increases. Our prediction results using time-series techniques (in Sec. 5.2) indicate that the prediction error can exceed 20% as the prediction window size reaches 6. *Second*, blindly increasing the time window size can significantly increase the number of samples of *Tetris*, which requires noticeable computation overhead to obtain container scheduling plans. To the best of our knowledge, *Tetris* is the first attempt to achieve the *long-term* optimization of container scheduling to *circumvent as many as possible invalid migrations*, by jointly optimizing the cluster load balancing and container migration cost over a certain sliding time window. Our main contributions are summarized as follows.

▷ *First*, we build a discrete-time dynamic model for shared containerized clusters to capture the time-varying resource consumption of containers and the corresponding mapping of containers on servers (Sec. 3). Based on such a model, we further devise a cost function of container scheduling *over a time window* and formulate our *long-term* workload scheduling optimization problem based on MPC, by jointly considering the cluster load imbalance degree and migration cost of containers.

▷ *Second*, we design *Tetris* to achieve *long-term* scheduling optimization for cluster load balancing (Sec. 4). Specifically, the *Container Resource Predictor* in *Tetris* first accurately predicts the container resource consumption (Sec. 4.1). The *MPC-based Container Scheduler* then leverages the Monte Carlo method [16] to judiciously identify the container scheduling decisions for each timeslot, by minimizing our formulated cost function of container scheduling (Sec. 4.2). To particularly reduce the complexity of *Tetris* container scheduling strategy, we calculate the thresholds of server load imbalance degree to classify the migration source and destination servers in *Tetris*.

▷ *Finally*, we implement a prototype of *Tetris* (https://github.com/icloud-ecnu/Tetris) based on K8s. We evaluate the effectiveness and runtime overhead of *Tetris* with prototype experiments on 10 EC2 instances (*i.e.*, 60 containers) and large-scale simulations driven by Alibaba cluster trace v2018 (Sec. 5). Compared with the conventional scheduling strategy (*i.e.*, Sandpiper [8]) and the state-of-the-art RL-based method (*i.e.*, Metis$^+$, a modified version of Metis [3]), *Tetris* is able to improve the cluster load balancing degree by up to 77.8% while cutting down the migration cost by up to 79.5%, yet with acceptable runtime overhead.

2 Background and Motivation

2.1 Long-running Containerized Workloads

Large-scale shared production clusters often host many long-running containers in response to latency-critical user requests [17]. Taking Alibaba as an example, its online services are mainly long-running and *stateful* containerized applications, such as online shopping, database, and machine learning [18]. As shown in Fig. 2, more than half of the containers are executed over 175 hours in the Alibaba production clusters. Additionally, the stateful (including partially stateful) workloads account for over 50% in the latest measurement study in Microsoft production clusters [19]. Due to the unpredictability of user requests, such long-running workloads place diverse demands on multi-dimensional container resources including CPU, GPU, memory, network and disk I/O, which are characterized by dynamic variability and uncertainty in the resource consumption [20]. Accordingly, the co-location of containerized workloads can lead to contention for shared resources, which can easily cause performance interference and resource wastage in the shared clusters.

Based on the above, enabling adequate (re)scheduling of long-running containerized workloads is essential to cluster load balancing. In shared production clusters, K8s achieves load balancing among servers by carrying out container *rescheduling*[1] (*i.e.*, container migrations [21]). Currently, CRIU (Checkpoint/Restore In Userspace)[2] supports the *pre-copy*-based *live* migration of containers [22]. To simplify the container migration process, we *simulate* it with the

1. Descheduler for Kubernetes: https://github.com/kubernetes-sigs/descheduler
2. CRIU: https://criu.org/Main_Page

This article has been accepted for publication in IEEE Transactions on Services 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/TSC.2024.3442544

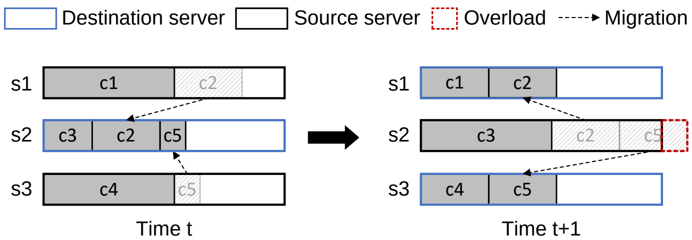


Fig. 3: Illustration of invalid migration. Containers $c2$ and $c5$ are migrated back and forth among servers $s1$, $s2$, and $s3$.

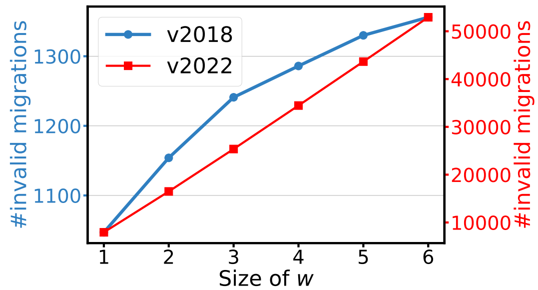


Fig. 4: Number of invalid migrations within a time window size varying from 1 to 6 hours for $67,437$ containers in Alibaba cluster trace v2018 and $676,938$ containers in Alibaba cluster trace v2022.

following three steps: *i.e.,* stopping containerized services on the source server first, then iteratively replicating the memory dirty pages to restore services, and finally restarting the services on the destination server. In general, the container scheduling inevitably brings a noticeable amount of migration cost, which requires to be considered during the scheduling of containers, especially for *stateful* workloads.

2.2 Invalid Migrations of Long-Running Workloads

While many existing scheduling policies (*e.g.,* Sandpiper [8]) perform well in load balancing in shared containerized clusters, they usually focus on the *short-term* benefits of container scheduling. Such a *short-term* optimization of container scheduling policy can easily make *invalid migration* decisions for containers, resulting in unpredictable migration cost and potential SLO violations.

Specifically, we take a cluster of three servers hosting five containers illustrated in Fig. 3 as an example. The cluster scheduler (*e.g.,* the default K8s scheduler) first migrates container $c2$ and $c5$ from server $s1$ and $s3$, respectively, to server $s2$, in order to obtain a *temporary* benefits of load balancing at time t. Unfortunately, server $s2$ becomes overloaded as the resource consumption of three containers (*i.e.,* $c3$, $c2$, $c5$) dramatically increases, which triggers two container migrations (*i.e.,* container $c2$, $c5$) from server $s2$ to $s1$ and $s3$, respectively. In such a case, naive scheduling policies are likely to make long-running containers migrated back and forth among servers, thereby causing heavy and unnecessary migration cost to containers. Accordingly, we formally define *invalid migrations* as in Definition 1.

Definition 1. *If a container is migrated back and forth among servers over a time window* $[t, t+w]$*, we define such container migrations as invalid migrations within a time window size* w*.*

Moreover, we validate the prevalence of invalid migrations with Sandpiper using both Alibaba cluster traces v2018 (*i.e.,* an 8-day period) and v2022 (*i.e.,* a 13-day period) [9]. As shown in Fig. 4, the number of invalid migrations increases as the scheduling time window increases. Specifically, the cumulative number of invalid migrations exceeds $1,200$ and $30,000$ for the v2018 and v2022 traces, respectively, as w is set as 3 hours. The number reaches over $50,000$, accounting for over 75% of container migrations for the v2022 trace when w increases to 6 hours. To further illustrate the performance impact of invalid migrations, we conduct experiments on a cluster of 10 servers (*i.e.,* 60 containers) deployed with Apache Tomcat Web server (partially stateful workloads). Our experiment results reveal that 61.7% of the containers are affected by invalid migrations. The number of requests processed by a migrated container can be reduced by up to 99.8% (74.0% on average), severely degrading the quality of long-running Web service. Therefore, it is essential to design a *long-term* container scheduling strategy to alleviate invalid migrations by explicitly considering the future resource consumption of containers.

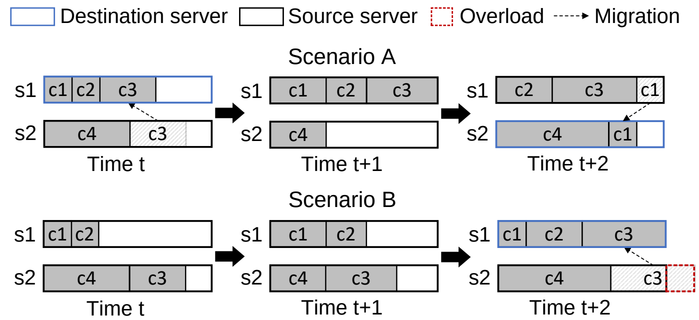


Fig. 5: An illustrative example of MPC-based container scheduling (our proposed *Tetris* in scenario A), as compared with RL-based container scheduling (Metis$^+$ [3] in scenario B).

2.3 An Illustrative Example

To avoid invalid migrations, we design *Tetris* in Sec. 4, a *simple yet effective* container scheduling strategy that leverages the MPC approach to proactively migrate long-running workloads for achieving the cluster load balancing. In particular, MPC adopts a compromise strategy that allows the current timeslot to be optimized while taking *finite future timeslots* into account [23]. To illustrate how *Tetris* works, we show a motivation example in Fig. 5 by comparing *Tetris* with an RL-based scheduling method (*i.e.,* Metis$^+$, a modified version of Metis [3] which will be introduced in Sec. 5.1). Though Metis$^+$ greedily achieves the optimal load balancing degree and minimizes the migration cost over the *entire* time period of $[t, t+2]$, it can still overload server $s2$ at time $t+2$ and trigger the migration of container $c3$ from server $s2$ to $s1$. In contrast, our MPC approach (*i.e., Tetris*, with the window size set as 2) jointly optimizes the load balancing degree and migration cost for two *sliding* timeslots (*i.e.,* $[t, t+1]$, and $[t+1, t+2]$) and guarantees SLOs of all containerized workloads. Accordingly, the RL-based method can optimize the scheduling objective *over the entire time period* at the cost of overloading a certain number of containers, while *Tetris* achieves the *long-term* optimization for container scheduling *within a certain sliding time window*. In addition, the RL-based method has the following problems such as requiring repeated training on a large amount of high-quality data samples and lacking

This article has been accepted for publication in IEEE Transactions on Services 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/TSC.2024.3442544

TABLE 1: Key notations in our discrete-time dynamic model at time t.

Notation	Definition
$\mathcal{M}, \mathcal{N}$	Sets of servers and containers
W	Time window size
$C_b(t)$	Load imbalance degree of the cluster
$C_m(t)$	Container migration cost of the cluster
$cpu^k(t)$	CPU consumption of a container k
$mem^k(t)$	Memory consumption of a container k
$\overline{CPU(t)}$	Average CPU consumption of a server
$\overline{MEM(t)}$	Average memory consumption of a server
α	Normalized model coefficient of migration cost to load imbalance degree
β	Normalized model coefficient of memory resources to CPU resources
γ, δ	Linear model coefficients of container migration cost in terms of the memory size
$x_i^k(t)$	Indicator if a container k is on a server i
$m^k(t)$	Indicator whether a container k is migrated

interpretability as well as poor scalability [3], which will be further validated in Sec. 5.4.

Summary. Avoiding invalid migrations is critical to container scheduling, and such invalid migrations are mainly caused by the *short-term* (*e.g.,* the current or upcoming timeslot) optimization of container scheduling. Moreover, greedily optimizing container scheduling over the entire time period (*i.e., infinite future*) is likely to cause unexpected SLO violations. Accordingly, there is a compelling need to design a *long-term* container scheduling strategy, by jointly optimizing the cluster load balancing and migration cost of containers within a certain sliding time window.

3 Model and Problem Formulation

In this section, we first build a discrete-time dynamic model to capture the load imbalance degree of clusters and the migration cost of containers. Next, we formulate a container scheduling optimization problem based on our dynamic model. The key model notations are summarized in Table 1.

3.1 Discrete-Time Dynamic Model

We consider a containerized cluster with a set of servers $\mathcal{M}$ hosting a set of containers $\mathcal{N}$. We assume the time t is discrete and slotted, where $t = \{t_0, t_1, ..., t_W\}$ and W is the time window size. For each container $k \in \mathcal{N}$, its CPU and memory resource consumption at each time t is recorded as $cpu^k(t)$ and $mem^k(t)$, respectively. For each server $i \in \mathcal{M}$, $CPU_i(t)$ and $MEM_i(t)$ represent its total CPU and total memory resource consumption at time t, respectively.

Cluster load imbalance degree. We use the variance of resource consumption of all cluster servers to represent the load imbalance degree of the cluster. A larger degree value indicates a severer load imbalance situation. By taking CPU and memory resources as an example, the cluster load imbalance degree is represented by calculating the weighted sum of the load imbalance degrees of each resource of servers, which is given by

$$C_b(t) = \frac{1}{|\mathcal{M}|-1}\sum_{i\in\mathcal{M}}\Bigg(\Big(\sum_{k\in\mathcal{N}} x_i^k(t)\cdot cpu^k(t) - \overline{CPU(t)}\Big)^2 + \beta\cdot\Big(\sum_{k\in\mathcal{N}} x_i^k(t)\cdot mem^k(t) - \overline{MEM(t)}\Big)^2\Bigg), \quad (1)$$

where $\beta \in [0,1]$ denotes the normalized parameter of memory resources to CPU resources. In practice, β can be empirically determined as the square of the ratio of CPU to memory resource consumption of workloads. $x_i^k(t)$ denotes whether the container k is hosted on the server i.

$$x_i^k(t) = \begin{cases} 1, & \text{if a container } k \text{ runs on a server } i, \\ 0, & \text{otherwise.}\end{cases}$$

Then, we proceed to denote the average CPU resource consumption $\overline{CPU(t)} = \frac{1}{|\mathcal{M}|}\sum_{k\in\mathcal{N}} cpu^k(t)$ and average memory resource consumption $\overline{MEM(t)} = \frac{1}{|\mathcal{M}|}\sum_{k\in\mathcal{N}} mem^k(t)$ of a server in the cluster.

Container migration cost. We use the sum of unit cost of the migrated containers $C_{mig}^k(t)$ to denote the migration cost $C_m(t)$, which is formulated as

$$C_m(t) = \sum_{k\in\mathcal{N}} C_{mig}^k(t)\cdot m^k(t), \quad (2)$$

where $m^k(t) = \sum_{i\in\mathcal{M}} x_i^k(t)\cdot\big(1 - x_i^k(t-1)\big)$ indicates whether a container k is migrated from a source server i at time t. As elaborated in Sec. 2.1, the migration cost of containers is *roughly linear* to their memory footprint $mem^k(t)$ [24]. Accordingly, $C_{mig}^k(t)$ can be given by

$$C_{mig}^k(t) = \delta + \gamma\cdot mem^k(t), \quad (3)$$

where γ and δ are linear model coefficients, which are empirical and constant values. We elaborate the configuration process of model coefficients in Appendix C.

To sum up, we further formulate the *cost function $C(t)$ of container scheduling at each timeslot*, by combining the cluster load imbalance degree $C_b(t)$ and the migration cost $C_m(t)$ as below,

$$C(t) = C_b(t) + \alpha\cdot C_m(t), \quad (4)$$

where $\alpha \in [0,1]$ denotes the normalized parameter of migration cost $C_m(t)$ to cluster load imbalance degree $C_b(t)$, which can practically be obtained by the trace analysis. By incorporating $\overline{CPU(t)}$ and $\overline{MEM(t)}$ into Eq. (1), and Eq. (3) and $m^k(t)$ into Eq. (2), we yield $C_b(t)$ and $C_m(t)$, respectively, which are formulated as

$$C_b(t) = \frac{2}{|\mathcal{M}|-1}\sum_{i\in\mathcal{M}}\sum_{k,l\in\mathcal{N}} x_i^k(t)x_i^l(t)\cdot\Big(cpu^k(t)cpu^l(t) + \beta\cdot mem^k(t)mem^l(t)\Big) + C_1, \quad (5)$$

$$C_m(t) = C_2 - \sum_{k\in\mathcal{N}}\sum_{i\in\mathcal{M}}\big(\delta + \gamma\cdot mem^k(t)\big)\cdot x_i^k(t)x_i^k(t-1). \quad (6)$$

Given a containerized cluster and workloads at time t, $C_1 = -\frac{|\mathcal{M}|\cdot\left(\overline{CPU(t)}^2 + \beta\cdot\overline{MEM(t)}^2\right)}{|\mathcal{M}|-1} - \frac{\sum_{k\in\mathcal{N}}\left(cpu^k(t)^2 + \beta\cdot mem^k(t)^2\right)}{|\mathcal{M}|-1}$ and $C_2 = \delta\cdot|\mathcal{N}| + \gamma\cdot\sum_{k\in\mathcal{N}} mem^k(t)$ are both *constant*

This article has been accepted for publication in IEEE Transactions on Services 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/TSC.2024.3442544

values. The mathematical derivations can be found in Appendix A. By analyzing the formulations above, Eq. (5) indicates that the cluster load imbalance degree is determined by the sum of the pairwise multiplications between the resource consumption of containers hosted on each server. Eq. (6) implies that the migration cost across the cluster can be determined by the negative sum of migration costs of the migrated containers.

3.2 Workload Scheduling Optimization Problem over a Time Window

Based on our discrete-time dynamic model above, we proceed to formulate the *long-term* optimization problem of container scheduling based on MPC. At each time t, MPC leverages the predicted container resource consumption (*i.e.*, $cpu^k(t)$, $mem^k(t)$) to make scheduling decisions (*i.e.*, judiciously deciding $x_i^k(t)$) to minimize the cost function $C(t)$ over the time window (as in Eq. (7)). We assume the scheduling within the time window $t \in [t_1, t_W]$, and our optimization problem can be formulated as

$$\min_{x_i^k(t)} \quad \sum_{t=t_1}^{t_W} C(t) \tag{7}$$

$$\text{s.t.} \quad \sum_{i \in \mathcal{M}} x_i^k(t) = 1, \quad \forall k \in \mathcal{N} \tag{8}$$

$$\sum_{k \in \mathcal{N}} x_i^k(t) \cdot cpu^k(t) \leq CPU_i^{cap}(t), \quad \forall i \in \mathcal{M} \tag{9}$$

$$\sum_{k \in \mathcal{N}} x_i^k(t) \cdot mem^k(t) \leq MEM_i^{cap}(t), \forall i \in \mathcal{M} \tag{10}$$

where the hard constraints in our optimization problem are as follows. Constraint (8) limits each container to be hosted only on one server per time to avoid scheduling conflicts. Constraints (9)-(10) ensure the total CPU and memory resource consumption on each server cannot exceed the server resource capacity (*i.e.*, $CPU_i^{cap}(t)$ and $MEM_i^{cap}(t)$).

Problem analysis. For each timeslot, the problem defined in Eq. (7) can be easily reduced to a *multiprocessor scheduling problem (MSP)* [25]. MSP is considered as finding a schedule for multiple tasks to be executed on a multiprocessor system at different timeslots, with the aim of minimizing the completion time. Such a scheduling problem is known to be NP-hard [26]. Moreover, Eq. (7) is a multi-objective optimization problem that jointly considers minimizing the migration cost and the cluster load imbalance degree, which makes it hard to solve. As a result, our optimization problem turns out to be harder than MSP. In the upcoming section, we turn to leveraging the Monte Carlo method [16] to solve our long-term workload scheduling optimization problem.

4 *Tetris* DESIGN

Based on our discrete-time dynamic model defined in Sec. 3, this section designs *Tetris*, including a container resource predictor (Sec. 4.1) and an MPC-based container scheduler (Sec. 4.2), with the aim of jointly optimizing the cluster load balancing degree and container migration cost over a certain time window.

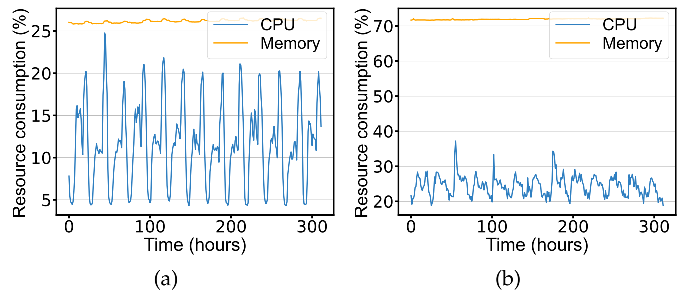


Fig. 6: Resource consumption of containers with respect to the physical machine resources over time: (a) a periodic container ($id = 404$), and (b) an aperiodic container ($id = 22$) in Alibaba cluster trace v2022.

4.1 Predicting Container Resource Consumption

We first classify workloads as *periodic* and *aperiodic* containers shown in Fig. 6, as most containerized workloads show a diurnal pattern [27]. Specifically, we first obtain the *frequency* of container resource consumption values and calculate its corresponding *periodicity*. With such a value, we then slice the container resource consumption into several segments. We finally calculate the Pearson correlation coefficients between every two consecutive segments and decide whether such coefficients exceed a given periodicity threshold thr_{period} (*e.g.*, 0.85). If the correlation coefficients exceed thr_{period}, we consider such a container as *periodic*. Otherwise, we consider the container as *aperiodic*.

Next, we proceed to predict the resource consumption for such two types of containers separately. Specifically, we adopt ARIMA [28] to predict the resource consumption of *periodic* containers. We mainly obtain three key parameters (*i.e.*, the autoregressive process of order p, the moving average process of order q, and the difference order d) of the ARIMA model. In addition, we leverage a simple yet effective LSTM [29] model to predict the resource consumption of *aperiodic* containers, rather than a more sophisticated attention-based model [30]. We adopt a 4-layer stacked LSTM model with 50 neurons per layer and add a *Dropout* layer between every two LSTM layers to reduce overfitting. To alleviate the impact of error propagation, we directly conduct the multistep-ahead predictions and set the number of prediction steps as W. In particular, we use the *historical* container resource consumption to train the *shared* ARIMA and LSTM models for newly-launched containers. To improve the prediction accuracy, we further leverage incremental learning [31] to update a *personalized* model for each container periodically.

4.2 MPC-based Container Scheduling

We design *Tetris* in Alg. 1 by leveraging the MPC approach to make container scheduling decisions over the time window W. Specifically, we use a $|\mathcal{M}| \times |\mathcal{N}|$ matrix $\mathbf{X}_t$ of $x_i^k(t)$ ($i \in \mathcal{M}, k \in \mathcal{N}$) to denote the mapping of containers to servers at time $t \in [t_0, t_W]$. According to MPC, we first solve the scheduling optimization problem in Sec. 3.2 to obtain $\mathbf{X}_t, \forall t \in [t_1, t_W]$, at the current time t_0. Then, we proactively perform the container scheduling decision $\mathbf{X}_{t_1}$ for the first timeslot t_1. After the containers are scheduled to migration destination servers at the "current" time t_1, we continue solving the scheduling optimization problem and then perform the container scheduling decisions for the

This article has been accepted for publication in IEEE Transactions on Services 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/TSC.2024.3442544

Algorithm 1: *Tetris*: Proactive MPC-based container scheduling algorithm for long-term load balancing.

Input: Current mapping $\mathbf{X}_{t_0}$ of container set $\mathcal{N}$ to server set $\mathcal{M}$, time window size W, number of samples Z, number of trials K.

1 **for** *all* $z \in [1, Z]$ **do**
2 **for** *all* $t \in [t_1, t_W]$ **do**
  `// Phase I: server classification`
3   **Initialize:** the load threshold of migration source servers $load_{src}(t) \leftarrow$ Eq. (12) and that of destination servers $load_{dest}(t) \leftarrow$ Eq. (13), the set of migration source/destination servers $\mathcal{M}_{src} \leftarrow \phi, \mathcal{M}_{dest} \leftarrow \phi$;
4   **for** *each server* $i \in \mathcal{M}$ **do**
5   Calculate $load_i(t) \leftarrow$ Eq. (11);
6   **if** $load_i(t) > load_{src}(t)$ **then**
7   Put server i in the set $\mathcal{M}_{src}$;
8   **if** $load_i(t) < load_{dest}(t)$ **then**
9   Put server i in the set $\mathcal{M}_{dest}$;
  `// Phase II: container scheduling`
10   **for** *all* $k \in [1, K]$ **do**
11   Obtain the set of migrated containers $\mathcal{N}_{mig} \leftarrow$ sampling containers from $\mathcal{M}_{src}$;
12   **for** *each container* $c \in \mathcal{N}_{mig}$ **do**
13   **for** *each server* $i \in \mathcal{M}_{dest}$ **do**
14   Calculate the container scheduling cost $C(t) \leftarrow$ Eq. (4), if container c is migrated to server i;
15   Update $\mathbf{X}_t$ by migrating container c to the server i with the smallest $C(t)$;
16   **if** $\mathbf{X}_t$ *satisfies the Constraints (8) – (10)* **then**
17   **break**;
18   Update $\mathbf{X}_t \leftarrow \mathbf{X}_{t-1}$; `// ` $\mathbf{X}_t$ ` cannot satisfy scheduling constraints`
19 Calculate the overall container scheduling cost over the time window W as $cost_z \leftarrow$ Eq. (7) with $\mathbf{X}_t$ for each sample z;
20 Obtain $\mathbf{X}_t^{min} \leftarrow \mathbf{X}_t$ with the minimum $cost_z$ among all Z samples;
21 **return:** Container scheduling decisions for the first timeslot (*i.e.*, $\mathbf{X}_{t_1}^{min}$).

"first" timeslot t_2. Accordingly, *Tetris* can be *periodically* (*e.g.*, for several minutes or hours) executed at each timeslot to maintain the load balancing of the containerized cluster.

In more detail, we solve the scheduling optimization problem using the Monte Carlo method [16] as discussed in Sec. 3.2. We take Z *sample* solutions in total, and each sample solution includes a set of container scheduling decisions $\mathbf{X}_t$ over the time window $t \in [t_1, t_W]$ (lines 1 to 2). To obtain the container scheduling decisions for each timeslot t, we design two phases in Alg. 1 including *server classification* (lines 3 to 9) and *container scheduling* (lines 10 to 18), which are elaborated as follows.

Phase I: server classification. We classify the cluster servers into three categories, *i.e.*, migration source servers and destination servers as well as the remaining servers (lines 3 to 9). Based on the cluster load imbalance degree (*i.e.*, Eq. (5)) defined in our dynamic model, we can obtain the simplified *load imbalance degree* (*i.e.*, $load_i(t)$) for *each server* i given by

$$load_i(t) = \sum_{k,l \in \mathcal{N}} x_i^k(t) x_i^l(t) \cdot \big(cpu^k(t) cpu^l(t) + \beta \cdot mem^k(t) mem^l(t)\big). \tag{11}$$

Obviously, the CPU and memory resource consumption of each server are $\overline{CPU(t)}$ and $\overline{MEM(t)}$, respectively, when the cluster reaches the *"ideal" load-balanced state*. As the load imbalance degree $load_i(t)$ of each server i can vary with the hosted containers and their resource consumption, we obtain the load threshold of migration source servers $load_{src}(t)$ and that of destination servers $load_{dest}(t)$ as in Theorem 1. In particular, *Tetris* can *tolerate moderate prediction errors* of container resource consumption for classifying migration source and destination servers.

Theorem 1. *Given a server hosting a set of containers with the average CPU and memory resource consumption (i.e., $\overline{CPU(t)}$ and $\overline{MEM(t)}$ at time t, we formulate the two load thresholds $load_{src}(t)$ and $load_{dest}(t)$ as*

$$load_{src}(t) = \frac{|\mathcal{N}| - 1}{2|\mathcal{N}|} \cdot \Big(\overline{CPU(t)}^2 + \beta \cdot \overline{MEM(t)}^2\Big), \tag{12}$$

$$load_{dest}(t) = \frac{1}{2}\Big(\overline{CPU(t)}^2 + \beta \cdot \overline{MEM(t)}^2 - \sum_{k=1}^{n(t)} \big(cpu^k(t)^2 + \beta \cdot mem^k(t)^2\big)\Big). \tag{13}$$

By sorting containers in descending order with the CPU and memory resource consumption, $cpu^k(t)$ and $mem^k(t)$ represent the CPU and memory consumption of the k-th container at time t, respectively. $n(t)$ denotes the minimum number of containers which satisfies $\sum_{k=1}^{n(t)} cpu^k(t) \leq \overline{CPU(t)}$ and $\sum_{k=1}^{n(t)} mem^k(t) \leq \overline{MEM(t)}$.

Proof. The proof can be found in Appendix B. □

Phase II: container scheduling. To make Alg. 1 practical, we sample a set of containers to be migrated $\mathcal{N}_{mig}$ from the set of migration source servers $\mathcal{M}_{src}$. To achieve extensive coverage of possibilities with a small number of samples, we adopt Latin Hypercube Sampling (LHS) with a sampling rate of v per migration source server (line 11). For each container to be migrated, we further select the migration destination server in $\mathcal{M}_{dest}$ with the smallest container scheduling cost and then update $\mathbf{X}_t$ (lines 12 to 15). Considering the randomness of sampling, the obtained container scheduling decisions $\mathbf{X}_t$ can violate constraints (8) – (10). If no violations occur, $\mathbf{X}_t$ is valid and we continue the container scheduling process for the next timeslot $t+1$. Otherwise, we require re-sampling $\mathcal{N}_{mig}$ and obtain another $\mathbf{X}_t$, until K trials are finished (line 10) or the scheduling constraints are satisfied (lines 16 to 17).

Finally, we iteratively calculate the overall container scheduling cost over the time window W for each sample z. We choose the container scheduling decisions $\mathbf{X}_t^{min}$ with the minimized scheduling cost among Z samples. According to

This article has been accepted for publication in IEEE Transactions on Services 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/TSC.2024.3442544

MPC, we only perform the container scheduling decisions $\mathbf{X}_{t_1}^{min}$ for the first timeslot t_1 (lines 19 to 21).

Determining parameters in *Tetris*. We obtain three key parameters (*i.e.*, v, W, Z) in Alg. 1 as below. *First*, we set the sampling rate v for migration containers as the minimum of two ratios (*i.e.*, one is the ratio of migration source servers with the CPU resource consumption exceeding $\overline{CPU(t)}$, and the other is the server ratio with the memory resource consumption exceeding $\overline{MEM(t)}$). *Second*, we configure the time window size W as the minimum of two values (*i.e.*, W_1, W_2). W_1 is the maximum time window with the prediction error of container resource consumption below a threshold (*e.g.*, 20%). W_2 is the window size with the fastest growing number of invalid migrations (in Fig. 4). *Third* is to set the number of samples Z. To uniformly sample different types of containers, we perform k-means clustering on the CPU and memory resource consumption of containers. We simply set the cluster number k as the lower bound of Z.

Time complexity analysis. According to Alg. 1, the time complexity of *Tetris* is in the order of $\mathcal{O}(ZWK \cdot |\mathcal{N}||\mathcal{M}|)$. As the number of containers $|\mathcal{N}|$ and the number of servers $|\mathcal{M}|$ are both *far larger* than the time window size W and the number of trials K, the complexity can be reduced to $\mathcal{O}(Z \cdot |\mathcal{N}| \cdot |\mathcal{M}|)$. Accordingly, given a containerized cluster and workloads, the computation overhead of *Tetris* can be practically acceptable, which will be validated in Sec. 5.4.

Prototype implementation. We implement a prototype of *Tetris* based on K8s, with over 1,000 lines of Python and Linux Shell codes which are publicly available on GitHub[3]. In particular, we implement the container migration module of *Tetris* by converting container scheduling decisions into a series of K8s pod operations (*i.e.*, pod deletion and creation commands executed on migration source and destination servers). We plan to integrate the container live migration of CRIU into *Tetris* as our future work.

5 Performance Evaluation

In this section, we evaluate *Tetris* by conducting prototype experiments based on a 60-container K8s cluster in Amazon EC2 and complementary large-scale simulations driven by real-world Alibaba production cluster traces v2018 and v2022. We focus on answering the questions listed below.

- *Accuracy:* Can *Tetris* accurately predict the resource consumption of long-running containers? (Sec. 5.2)
- *Effectiveness:* Can our *Tetris* scheduling strategy jointly optimize the cluster load balancing degree and migration cost for long-running containers without incurring SLO violations? (Sec. 5.3)
- *Overhead:* How much runtime overhead does *Tetris* practically bring? (Sec. 5.4)

5.1 Experimental Setup

Cluster configurations and workloads. We carry out prototype experiments upon 10 m6a.large EC2 instances. Each instance is equipped with 2 vCPUs and 8 GB memory and initially hosting 6 containers. We select three representative containerized workloads including Apache Tomcat

3. https://github.com/icloud-ecnu/Tetris

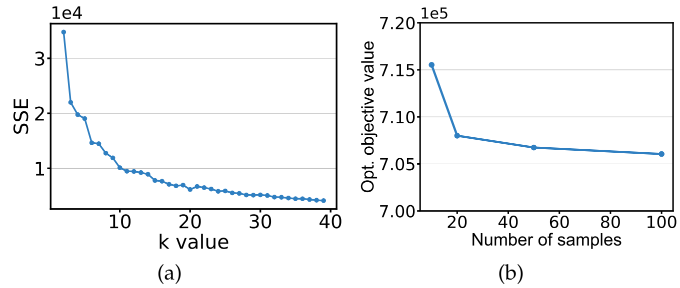


Fig. 7: (a) Sum of the squared errors (SSE) of k-means clustering on CPU and memory resource consumption of containers with different k values, and (b) the optimization objective value decreasing slowly from 20 samples in the Monte Carlo method.

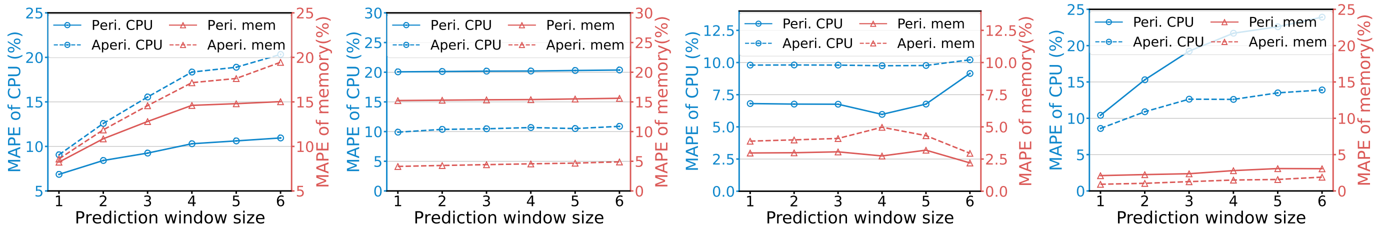
server[4], Redis[5], and ResNet50 [32] training. We use Apache-benchmark[6] and redis-benchmark[7] as the time-varying user requests for Tomcat Web server and Redis, respectively. We adopt CIFAR-10[8] as the dataset for ResNet50. We *randomly* deploy the three workloads on the 60 containers in our cluster. To obtain complementary insights, we build a trace-driven simulator based on a discrete-event simulation framework [33]. We adopt Alibaba cluster traces v2018 and v2022 [9] for the prediction of container resource consumption and large-scale simulations.

***Tetris* parameters and model coefficients.** We first configure four *Tetris* parameters. The number of samples Z is set as 20, as the container resource consumption can be clustered with 20 types and the optimization value converges at 20 samples shown in Fig. 7. The time window size W is set as 2 based on the prediction accuracy of container resource consumption. The number of trials K and the sampling ratio v of migration containers are empirically set as 10 and 40%, respectively. Second, we empirically set four model coefficients including α, β, γ, and δ as 0.004, 0.0025, 1, and 10, respectively. The detailed configuration process can be found in Appendix C.

Baselines and metrics. We evaluate *Tetris* against the conventional Sandpiper [8] and the state-of-the-art Metis$^+$ (*i.e.*, a modified version of Metis [3]) scheduling algorithms. To achieve load balancing, Sandpiper performs a greedy worst-fit algorithm to schedule containers according to the container index $volume$ calculated by the multi-dimensional resource consumption [8]. Metis$^+$ uses the method in *Tetris* to select migration source servers and migrated containers and leverages the negative value of Eq. (4) as the reward of RL to make container scheduling (*i.e.*, where to schedule) decisions. In particular, we use the *mean absolute percentage error (MAPE)* [34] to evaluate the efficacy of the *predictor* module of *Tetris*. Meanwhile, we adopt the *load imbalance degree* defined in Eq. (1) and the *migration cost* defined in Eq. (2) as well as the number of *SLO violations* to evaluate the efficacy of the *scheduler* module of *Tetris*. Due to the randomness of sampling in the Monte Carlo method and workload placement on containers, we illustrate the container scheduling performance with error bars of standard deviation by repeating experiments three times.

4. https://tomcat.apache.org/
5. https://redis.io/
6. https://httpd.apache.org/docs/2.4/programs/ab.html
7. https://redis.io/topics/benchmarks
8. https://www.cs.toronto.edu/~kriz/cifar.html

This article has been accepted for publication in IEEE Transactions on Services 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/TSC.2024.3442544



(a) ARIMA (trace v2018) (b) LSTM (trace v2018) (c) ARIMA (trace v2022) (d) LSTM (trace v2022)

Fig. 8: Average prediction accuracy of CPU and memory resource consumption of $67,437$ containers and $676,938$ containers in Alibaba cluster traces v2018 and v2022, respectively, by varying the prediction window size from 1 to 6.

TABLE 2: Number of containers with periodic or aperiodic CPU/memory resource consumption in Alibaba cluster trace v2018 and v2022.

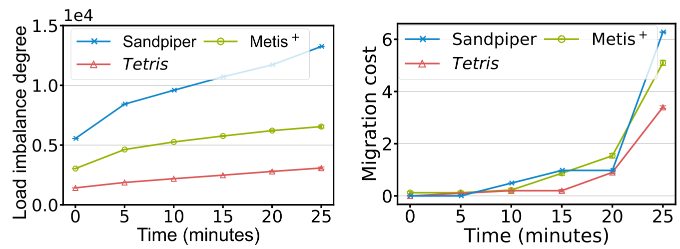
	Container type	CPU	Memory
v2018	#periodic containers	$49,962$	$40,501$
	#aperiodic containers	$17,475$	$26,936$
v2022	#periodic containers	$416,905$	$151,147$
	#aperiodic containers	$260,033$	$525,791$

5.2 Validating Container Resource Predictor in *Tetris*

We first classify the resource consumption of containers as periodic or aperiodic using the method elaborated in Sec. 4.1. As shown in Table 2, the periodic CPU and memory resource consumption accounts for $61.6\% - 74.1\%$ and $22.3\% - 60.1\%$ of containers, respectively, while the proportion of aperiodic CPU and memory resource consumption are only $25.9\% - 38.4\%$ and $39.9\% - 77.7\%$, respectively, for both Alibaba cluster traces v2018 and v2022.

We next examine the prediction accuracy of container resource consumption. We use 70% of the trace data to train the ARIMA and LSTM models in *Tetris*. As shown in Fig. 8, we *first* observe that ARIMA is more accurate than LSTM in predicting the resource consumption of periodic containers. This is because the moving average and autoregression in ARIMA accurately captures the periodic resource consumption of containers, with a small prediction error (*i.e.*, less than 15%). *Second*, LSTM achieves a more accurate prediction error (*i.e.*, 0.9% – 13.9%) for the resource consumption of aperiodic containers compared with ARIMA. This is mainly because ARIMA can easily be affected by abnormal spikes in aperiodic resource consumption, while LSTM can learn such large resource fluctuations through model training. Interestingly, LSTM achieves around $10.4\% - 23.9\%$ of prediction error for periodic containers as depicted in Fig. 8(b) and Fig. 8(d), simply because LSTM contains nonlinear activation functions and thus overfits the periodic data [35]. *Third*, the prediction error of container resource consumption can significantly increase from 0.9% to 23.9% with the time window size ranging from 1 to 6. That is the reason why *Tetris* controls the time window size W within 3 to maintain an acceptable prediction error (*i.e.*, within 20%) of container resource consumption.

We finally examine the prediction overhead of *Tetris*. Specifically, the average prediction time of ARIMA and LSTM are 0.25 seconds and 0.36 seconds, respectively, for each container. Though *Tetris* requires a shared model for all containers and a personalized model for each container, the average storage footprint of an ARIMA model and an LSTM model are just 556 KB and 908 KB, respectively. Such time and space overhead above for the prediction of container resource consumption is practically acceptable.



(a) (b)

Fig. 9: Performance comparison of *Tetris* with Sandpiper and Metis$^+$ strategies under K8s-based prototype experiments, in terms of the cumulative values of (a) cluster load imbalance degree and (b) migration cost of containers over time (*i.e.*, 5 timeslots with 5 minutes each).

5.3 Effectiveness of *Tetris*

Prototype experiments on Amazon EC2. We first evaluate the effectiveness of *Tetris* in a 60-container K8s cluster by setting the *timeslot*[9] as 5 minutes. As shown in Fig. 9, *Tetris* achieves a lower load imbalance degree by 74.4% – 77.8% and 53.0% – 59.6% compared with Sandpiper and Metis$^+$, respectively. Moreover, the migration cost with *Tetris* is 7.8% – 79.5% and 10.3% – 77.0% lower than that with Sandpiper and Metis$^+$, respectively. This is because (1) *Tetris* jointly optimizes the load imbalance degree and migration cost, while Sandpiper greedily migrates containers to alleviate the server hotspot without considering the negative impact of migration cost, which causes 37 invalid migrations in total. (2) Though Metis$^+$ leverages the RL method to explicitly consider the migration cost, it sacrifices the container performance for a certain number of timeslots to optimize container scheduling over the entire time period (*i.e.*, *infinite future*), bringing 12 invalid migrations for 5 timeslots. In contrast, *Tetris* continuously optimizes each timeslot (*i.e.*, 5 minutes) using a certain sliding time window.

To illustrate the effectiveness of *Tetris*, we further take a close look at the scheduling decisions made by the three strategies. As shown in Fig. 10, we observe that *Tetris* achieves the most load balancing than the other two strategies at both time 0 and time 1, by proactively migrating out four small containers (*i.e.*, $c3$ – $c6$) at time 0. In comparison, Sandpiper causes an invalid migration of container $c1$. As server $s1$ is overloaded at time 0 by setting the overload

9. The timeslot can be determined by the cluster administrator according to the cluster requirements.

This article has been accepted for publication in IEEE Transactions on Services 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/TSC.2024.3442544

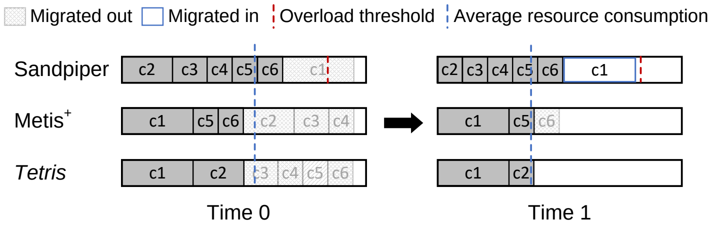


Fig. 10: Comparison of container scheduling decisions on server $s1$ of the 60-container K8s cluster in the timeslot $[0, 1]$ under *Tetris*, Sandpiper, and Metis$^+$ strategies.

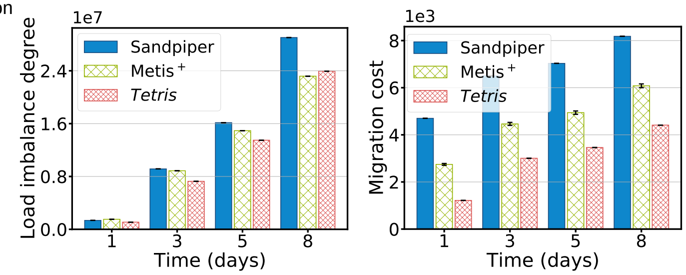
TABLE 3: Cumulative number of SLO violations over time in our prototype experiments on a 60-container K8s cluster.

Duration (timeslots)	0	1	2	3	4	5
Sandpiper	0	0	14	22	26	61
Metis$^+$	0	0	0	8	8	20
Tetris	0	0	0	0	0	0

threshold as 85%, Sandpiper chooses to migrate container $c1$ (*i.e.*, the container with the maximum $volume$) to server $s2$. It then migrates container $c1$ back to server $s1$ as server $s2$ is overloaded at time 1, resulting in poor load balancing and large migration cost. As for Metis$^+$, it migrates out containers (*i.e.*, $c2$ – $c4$) and container $c6$ at time 0 and time 1, respectively, because it adopts online training of the RL model, which has not been trained well enough over the initial time $[0, 1]$. In addition, Metis$^+$ mainly considers the optimization over the infinite future, which is likely to increase the load imbalance degree or migration cost over a certain time period (*e.g.*, time $[0, 1]$).

We proceed to examine whether *Tetris* can guarantee the SLO of workloads. For simplicity, we consider the containers hosted on the *overloaded* servers as SLO violations. As shown in Table 3, *Tetris* causes zero SLO violations, while Sandpiper and Metis$^+$ can cause up to 35 and 12 SLO violations, respectively, within one timeslot (*i.e.*, at time 5). The reason is that *Tetris* explicitly considers the hard constraints (*i.e.*, Constraints (9)–(10)) on the CPU and memory resource capacities of servers, guaranteeing the workload SLO for each timeslot. However, Sandpiper greedily migrates the container with the largest *volume* to the server with the lightest load. It does not check whether the server load exceeds the capacity on the migration destination server, causing unexpected SLO violations to containers. The RL method in Metis$^+$ is not designed to guarantee the scheduling constraints, so that it allows the occurrence of SLO violations over a certain time period to optimize container scheduling (*i.e.*, maximize the action reward) over the infinite future.

Large-scale simulations driven by Alibaba cluster trace. We next evaluate the effectiveness of *Tetris* with real-world trace-driven simulations, by setting the *timeslot* as 1 hour. As shown in Fig. 11, *Tetris* can reduce the load imbalance degree of the cluster by 9.7% – 28.6% compared with Sandpiper and Metis$^+$. Additionally, *Tetris* achieves the lowest migration cost, which is reduced by 27.4% – 74.1% than the other two strategies. Such results above are consistent with our prototype experiments. This is because *Tetris* co-optimizes the load balancing and migration cost to alleviate invalid migrations. In contrast, Sandpiper



(a) (b)

Fig. 11: Performance comparison of *Tetris* with Sandpiper and Metis$^+$ strategies under simulations on Alibaba cluster trace v2018, in terms of the cumulative values of (a) cluster load imbalance degree and (b) migration cost of containers over 8 days (the timeslot is 1 hour).

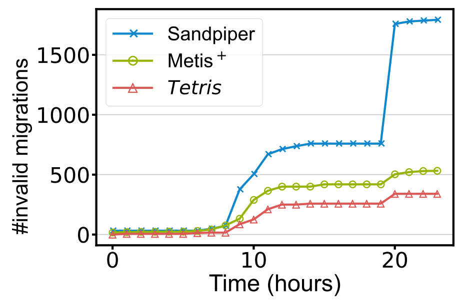


Fig. 12: Cumulative number of invalid migrations of *Tetris* compared with that of Sandpiper and Metis$^+$ strategies under simulations on a 24-hour Alibaba cluster trace v2018.

greedily alleviates the heavy server load using the worst-fit algorithm, resulting in many invalid migrations and a high growth rate of migration cost as depicted in Fig. 11(b). Though Metis$^+$ can optimize the load imbalance degree and migration cost over the infinite future, it achieves a worse load balancing degree in the early stage (*i.e.*, the first 5 days) and a better load degree in the late stage (*i.e.*, day 8) compared with *Tetris* as shown in Fig. 11(a). This is because it requires continuous online training until the RL model converges (after day 5), which is likely to cause a moderate number of invalid migrations in the early stages.

We further examine whether *Tetris* can alleviate invalid migrations. As shown in Fig. 12, *Tetris* reduces the number of invalid migrations by 59.4% – 81.3% and 32.4% – 78.9% as compared with Sandpiper and Metis$^+$, respectively. *Tetris* and Metis$^+$ can reduce the number of invalid migrations, simply because they both consider long-term co-optimization of load balancing and migration cost for container scheduling. In comparison, Sandpiper only considers short-term optimization of container scheduling, resulting in a significant number of invalid migrations. As Metis$^+$ achieves long-term container scheduling over the infinite future, which can cause more invalid migrations than *Tetris* over a certain number of timeslots. In particular, Sandpiper invokes a significant number of invalid migrations at time 20, because the resource consumption of containers varies greatly which overloads a number of servers at time 20.

Similar to our prototype experiments, we proceed to examine whether *Tetris* can guarantee the SLO of workloads in large-scale simulations. As shown in Table 4, we observe that *Tetris* does not cause any SLO violations for 8 days, while both Sandpiper and Metis$^+$ can cause up to $14,403$ and 918 SLO violations, respectively, which account for

This article has been accepted for publication in IEEE Transactions on Services 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/TSC.2024.3442544

TABLE 4: Cumulative number of SLO violations over time in a simulated $67,437$-container cluster driven by Alibaba cluster trace v2018.

Duration (days)	1	3	5	8
Sandpiper	881	3,512	12,248	14,403
Metis$^+$	0	220	220	918
Tetris	0	0	0	0

TABLE 5: Performance comparison of *Tetris* with various parameter configurations (*i.e.*, $[W, Z]$) on a 24-hour Alibaba cluster trace v2018, by jointly considering cluster load imbalance degree ($C_b(t)$) and migration cost of containers ($C_m(t)$). $[2, 20]$ particularly denotes the default *Tetris*.

$[W, Z]$	$[2, 20]$	$[4, 20]$	$[6, 20]$	$[2, 10]$	$[2, 30]$
$C_b(t)$	**59.8**	60.8	63.1	62.4	60.4
$C_m(t)$	**1.6**	2.0	3.5	2.8	1.7

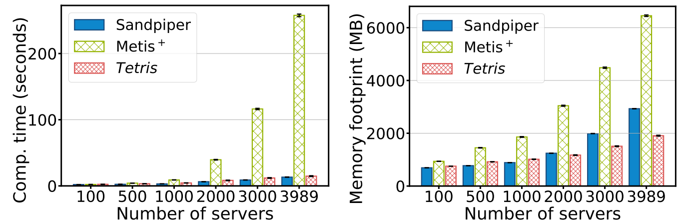
21.4% and 1.4% of the total number of containers. Our simulation results above are consistent with the prototype experiments, demonstrating the effectiveness of *Tetris* in guaranteeing workload SLOs.

We finally conduct a sensitivity analysis of two key *Tetris* parameters (*i.e.*, the window size W and the number of samples Z) in Alg. 1. As shown in Table 5, the default *Tetris* achieves the *lowest* container scheduling performance among various parameter configurations. Specifically, the performance metrics (*i.e.*, $C_b(t)$ and $C_m(t)$) both increase *moderately* as W varies from 2 to 6, simply because the prediction error of container resource consumption increases as shown in Fig. 8. Additionally, the performance of container scheduling first decreases and then stabilizes as Z increases from 10 to 30, which is consistent with our configuration analysis of Z in Fig. 7.

5.4 Runtime Overhead of *Tetris*

As shown in Fig. 13(a), we observe that the runtime overhead of the three strategies increases *quadratically* with the cluster scale, while Sandpiper and *Tetris* achieve much lower overhead than Metis$^+$. The rationale is that the search space of Metis$^+$ contains all possible mappings of containers to servers, while Sandpiper and *Tetris* only consider the overloaded servers (containers) and the containers on the migration source and destination servers, respectively. Though the time complexity of *Tetris* is in the order of $\mathcal{O}(|\mathcal{N}| \cdot |\mathcal{M}|)$ (as discussed in Sec. 4.2), the *quadratic term ratio* (obtained using polynomial regression) of *Tetris* is only $1.9e-07$, which is much smaller than that of Metis$^+$ (*i.e.*, $2.4e-05$) and Sandpiper (*i.e.*, $5.5e-07$). Similarly, *Tetris* consumes a much smaller amount of memory than Metis$^+$ and Sandpiper as shown in Fig. 13(b). This is because Metis$^+$ stores the policy of RL which involves all states and actions as well as the large parameters of neural networks. Sandpiper requires storing the $volumes$ of all containers and servers. In comparison, *Tetris* only needs to calculate the load imbalance degree of all servers and the migration cost of a subset of containers (*i.e.*, containers to be migrated).

In addition, we illustrate the computation overhead of three strategies over various time scales (*i.e.*, from 1 to 8 days) in Table 6. The computation time of *Tetris* is slightly longer than that of Sandpiper (*i.e.*, 0.7 – 4.8 minutes), while much shorter than that of Metis$^+$ (*i.e.*, 79.6 – 775.6 minutes) as the time scale increases. This is because Sandpiper triggers the fewest algorithm executions only when the resource hotspot occurs. Metis$^+$ greedily selects the largest reward action across all state spaces in each timeslot. Also, it requires online training to update the RL model. In contrast, *Tetris* periodically triggers the execution of Alg. 1 in each timeslot. Such time overhead is much smaller than Metis$^+$ per timeslot as depicted in Fig. 13(a). As a result, the runtime overhead of *Tetris* is practically acceptable.



(a) Computation time (b) Memory footprint

Fig. 13: Runtime overhead of *Tetris* compared with that of Sandpiper and Metis$^+$ strategies for each timeslot (*i.e.*, 1 hour) on a 24-hour Alibaba cluster trace v2018, by varying the number of servers from 100 to $3,989$ with each hosting 17 containers.

TABLE 6: Cumulative computation overhead (in minutes) of *Tetris*, Sandpiper and Metis$^+$ scheduling algorithms executed on a 8-day Alibaba cluster trace v2018.

Duration (days)	1	3	5	8
Sandpiper	4.5	17.6	28.5	46.7
Metis$^+$	86.1	333.6	503.2	827.1
Tetris	6.5	19.8	29.2	51.5

6 Related Work

Short-term optimization of workload scheduling. There have been many works devoted to scheduling VMs or containers by considering the *short-term* (*e.g.*, the current or upcoming timeslot) benefits. To predict the resource consumption of workloads, Dabbagh et al. [36] leverage the Wiener filter method for achieving VM consolidations [37], while Madu [30] designs a more complicated attention-based machine learning model to reduce workload latency. Bayesian optimization [38] and prediction-based vertical auto-scaling [39] have been proposed to improve cluster utilization and workload performance. Medea [2] further considers several constraints like affinity and cardinality of containers. To particularly optimize the initial placement of containers, Lv et al. [24] propose a communication-aware worst-fit decreasing algorithm and select the best server for satisfying the network SLO in K8s [40]. To achieve cluster load balancing, Sandpiper [8] performs the worst fitting algorithm using the server load index based on cluster resources. ATCM [41] schedules containers among the cloud servers to minimize the migration cost while maintaining the degree of load balance in a short time scale. Nevertheless, such *short-term* optimizations of workload scheduling above can inevitably cause invalid migrations as illustrated in Sec. 2.2. In contrast, *Tetris* leverages the MPC approach to achieve the *long-term* optimization of container scheduling and jointly optimize load balancing and migration cost for long-running workloads.

RL-based workload scheduling. Two recent works (*i.e.,* Metis [3], George [5]) design RL algorithms to obtain efficient initial placement plans for long-running containers, by modeling the reward as an indicator of container performance and constraint violations. To improve cluster utilization Mondal et al. [4] schedule time-varying workloads to servers using deep reinforcement learning (DRL). To consider the *long-term* optimization of workload scheduling, Megh [42] leverages an online RL-based VM migration method to minimize energy consumption and avoid SLO violations. A-SARSA [43] adopts an RL-based container horizontal scaling approach to adjust the number of actions corresponding to each state to avoid repeated scheduling. However, RL-based scheduling methods can cause unexpected SLO violations by considering the long-term optimization over the infinite future, as evidenced by Sec. 2.3 and Sec. 5.3. Additionally, RL methods have high time complexity and memory consumption even after the dimensionality reduction, which requires efforts to be deployed in large-scale clusters as shown in Sec. 5.4. To reduce the computation complexity to minutes, *Tetris* leverages the MPC approach to obtain a sub-optimal solution over a certain and sliding time window, which is sufficient for our requirements of *long-term* container scheduling in production clusters.

7 Conclusion and Future Work

This paper presents the design and implementation of *Tetris*, a proactive MPC-based container scheduling strategy for achieving long-term load balancing of clusters. By devising a discrete-time dynamic model of shared containerized clusters, *Tetris* judiciously identifies the container scheduling decisions (*i.e., which container to schedule* and *where to schedule*) for each timeslot using the Monte Carlo method, with the aim of jointly optimizing cluster load balancing and migration cost of containers. We implement a prototype of *Tetris* and conduct prototype experiments on Amazon EC2 and large-scale simulations driven by Alibaba cluster traces v2018 and v2022. Our experiment results demonstrate that *Tetris* can improve the cluster load balancing degree by up to 77.8%, while reducing the migration cost by up to 79.5% with acceptable runtime overhead, compared with the state-of-the-art container scheduling strategies.

As our future work, we plan to extend our discrete-time dynamic model by incorporating more types of server resources such as GPU, network and disk I/O resources. We also plan to incorporate CRIU into *Tetris* to support container live migrations and examine the effectiveness and scalability of *Tetris* in large-scale production clusters.

References

[1] M. Brooker, M. Danilov, C. Greenwood, and P. Piwonka, "On-demand Container Loading in AWS Lambda," in *Proc. of USENIX ATC*, Jul. 2023, pp. 315–328.

[2] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao, "Medea: Scheduling of Long Running Applications in Shared Production Clusters," in *Proc. of Eurosys*, Apr. 2018, pp. 1–13.

[3] L. Wang, Q. Weng, W. Wang, C. Chen, and B. Li, "Metis: Learning to Schedule Long-Running Applications in Shared Container Clusters at Scale," in *Proc. of IEEE SC*, Nov. 2020, pp. 1–17.

[4] S. S. Mondal, N. Sheoran, and S. Mitra, "Scheduling of Time-Varying Workloads Using Reinforcement Learning," in *Proc. of AAAI*, vol. 35, no. 10, Feb. 2021, pp. 9000–9008.

[5] S. Li, L. Wang, W. Wang, Y. Yu, and B. Li, "George: Learning to Place Long-Lived Containers in Large Clusters with Operation Constraints," in *Proc. of ACM SOCC*, Nov. 2021, pp. 258–272.

[6] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-Scale Cluster Management at Google with Borg," in *Proc. of Eurosys*, Apr. 2015, pp. 1–17.

[7] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[8] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Sandpiper: Black-box and Gray-box Resource Management for Virtual Machines," *Computer Networks*, vol. 53, no. 17, pp. 2923–2938, 2009.

[9] Alibaba. (2024, Jul.) Alibaba Cluster Trace Program. [Online]. Available: https://github.com/alibaba/clusterdata/

[10] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-Aware Resource Allocation Heuristics for Efficient Management of Data Centers for Cloud Computing," *Future Generation Computer Systems*, vol. 28, no. 5, pp. 755–768, 2012.

[11] M. Xu, W. Tian, and R. Buyya, "A Survey on Load Balancing Algorithms for Virtual Machines Placement in Cloud Computing," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 12, p. e4123, 2017.

[12] H. Qiu, W. Mao, C. Wang, H. Franke, A. Youssef, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "AWARE: Automate Workload Autoscaling with Reinforcement Learning in Production Cloud Systems," in *Proc. of USENIX ATC*, Jul. 2023, pp. 387–402.

[13] M. Gaggero and L. Caviglione, "Model Predictive Control for the Placement of Virtual Machines in Cloud Computing Applications," in *Proc. of ACC*, Jul. 2016, pp. 1987–1992.

[14] ——, "Model Predictive Control for Energy-Efficient, Quality-Aware, and Secure Virtual Machine Placement," *IEEE Transactions on Automation Science and Engineering*, vol. 16, no. 1, pp. 420–432, 2018.

[15] C. E. Garcia, D. M. Prett, and M. Morari, "Model Predictive Control: Theory and Practice – A Survey," *Automatica*, vol. 25, no. 3, pp. 335–348, 1989.

[16] N. Metropolis and S. Ulam, "The Monte Carlo Method," *Journal of the American Statistical Association*, vol. 44, no. 247, pp. 335–341, 1949.

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

[18] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces," in *Proc. of IEEE/ACM IWQoS*, Jun. 2019, pp. 1–10.

[19] A. Parayil, J. Zhang, X. Qin, igo Goiri, L. Huang, T. Zhu, and C. Bansal, "Towards Cloud Efficiency with Large-scale Workload Characterization," *arXiv preprint arXiv:2405.07250*, 2024.

[20] M. Xu, C. Song, H. Wu, S. S. Gill, K. Ye, and C. Xu, "esDNN: Deep Neural Network based Multivariate Workload Prediction in Cloud Computing Environments," *ACM Transactions on Internet Technology*, vol. 22, no. 3, pp. 1–24, 2022.

[21] S. Deng, H. Zhao, B. Huang, C. Zhang, F. Chen, Y. Deng, J. Yin, S. Dustdar, and A. Y. Zomaya, "Cloud-Native Computing: A Survey from the Perspective of Services," *Proceedings of the IEEE*, vol. 112, no. 1, pp. 12–46, 2024.

[22] J. Thiede, N. Bashir, D. Irwin, and P. Shenoy, "Carbon Containers: A System-level Facility for Managing Application-level Carbon Emissions," in *Proc. of ACM SOCC*, Oct. 2023, pp. 17–31.

[23] Y. Zhang, L. Jiao, J. Yan, and X. Lin, "Dynamic Service Placement for Virtual Reality Group Gaming on Mobile Edge Cloudlets," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 8, pp. 1881–1897, 2019.

[24] L. Lv, Y. Zhang, Y. Li, K. Xu, D. Wang, W. Wang, M. Li, X. Cao, and Q. Liang, "Communication-Aware Container Placement and Reassignment in Large-Scale Internet Data Centers," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 540–555, 2019.

This article has been accepted for publication in IEEE Transactions on Services 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/TSC.2024.3442544

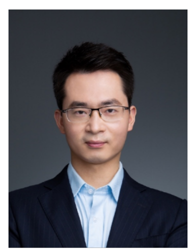
[25] E. S. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 2, pp. 113–120, 1994.
[26] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Hardness*. San Fransisco: W. H. Freeman, 1979.
[27] G. Christofidi, K. Papaioannou, and T. D. Doudali, "Is Machine Learning Necessary for Cloud Resource Usage Forecasting?" in *Proc. of ACM SOCC*, Oct. 2023, pp. 544–554.
[28] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control*. John Wiley & Sons, 2015.
[29] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[30] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, and C. Xu, "The Power of Prediction: Microservice Auto Scaling via Workload Learning," in *Proc. of ACM SOCC*, Nov. 2022, pp. 355–369.
[31] Z. Li and D. Hoiem, "Learning without Forgetting," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 12, pp. 2935–2947, 2017.
[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proc. of CVPR*, Jul. 2016, pp. 770–778.
[33] F. Li and B. Hu, "Deepjs: Job Scheduling Based on Deep Reinforcement Learning in Cloud Data Center," in *Proc. of ACM ICBDC*, May 2019, pp. 48–53.
[34] A. De Myttenaere, B. Golden, B. Le Grand, and F. Rossi, "Mean Absolute Percentage Error for Regression Models," *Neurocomputing*, vol. 192, pp. 38–48, 2016.
[35] D. Fan, H. Sun, J. Yao, K. Zhang, X. Yan, and Z. Sun, "Well Production Forecasting Based on ARIMA-LSTM Model Considering Manual Operations," *Energy*, vol. 220, pp. 1–13, 2021.
[36] M. Dabbagh, B. Hamdaoui, M. Guizani, and A. Rayes, "An Energy-Efficient VM Prediction and Migration Framework for Overcommitted Clouds," *IEEE Transactions on Cloud Computing*, vol. 6, no. 4, pp. 955–966, 2016.
[37] Z. Á. Mann, "Allocation of Virtual Machines in Cloud Data Centers–A Survey of Problem Models and Optimization Algorithms," *ACM Computing Surveys*, vol. 48, no. 1, pp. 1–34, 2015.
[38] C. Luo, B. Qiao, X. Chen, P. Zhao, R. Yao, H. Zhang, W. Wu, A. Zhou, and Q. Lin, "Intelligent Virtual Machine Provisioning in Cloud Computing," in *Proc. of IJCAI*, Jan. 2021, pp. 1495–1502.
[39] J. Zhu, R. Yang, X. Sun, T. Wo, C. Hu, H. Peng, J. Xiao, A. Y. Zomaya, and J. Xu, "QoS-Aware Co-Scheduling for Distributed Long-Running Applications on Shared Clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4818–4834, 2022.
[40] E. Kim, K. Lee, and C. Yoo, "Network SLO-Aware Container Scheduling in Kubernetes," *The Journal of Supercomputing*, vol. 79, no. 10, pp. 11 478–11 494, 2023.
[41] W. Zhang, L. Chen, J. Luo, and J. Liu, "A Two-Stage Container Management in the Cloud for Optimizing the Load Balancing and Migration Cost," *Future Generation Computer Systems*, vol. 135, pp. 303–314, 2022.
[42] D. Basu, X. Wang, Y. Hong, H. Chen, and S. Bressan, "Learn-as-you-go with Megh: Efficient Live Migration of Virtual Machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1786–1801, 2019.
[43] S. Zhang, T. Wu, M. Pan, C. Zhang, and Y. Yu, "A-SARSA: A Predictive Container Auto-Scaling Algorithm Based on Reinforcement Learning," in *Proc. of IEEE ICWS*, Oct. 2020, pp. 489–497.

Fei Xu received the B.S., M.E., and Ph.D. degrees in 2007, 2009, and 2014, respectively, all from the Huazhong University of Science and Technology (HUST), Wuhan, China. He is currently a full professor with the School of Computer Science and Technology, East China Normal University, Shanghai, China. He received Outstanding Doctoral Dissertation Award in Hubei province, China, and ACM Wuhan & Hubei Computer Society Doctoral Dissertation Award in 2015, and a recipient of the Best Student Paper Award of CCFSys 2023. His research interests include cloud computing and datacenter, and distributed machine learning systems.

Xiyue Shen received the master's degree with the School of Computer Science and Technology, East China Normal University, Shanghai, China. Her research interests focus on cloud computing and datacenter resource management.

Shuohao Lin is currently working toward the master's degree with the School of Computer Science and Technology, East China Normal University, Shanghai, China. His research interests focus on cloud computing and datacenter resource management.

Li Chen received the BEngr degree from the Department of Computer Science and Technology, Huazhong University of Science and Technology, China, in 2012 and the MASc degree from the Department of Electrical and Computer Engineering, University of Toronto, in 2014 and the PhD degree in computer science and engineering from the Department of Electrical and Computer Engineering, University of Toronto, in 2018. She is currently an assistant professor with the Department of Computer Science, School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, USA. Her research interests include big data analytics systems, cloud computing, datacenter networking, and resource allocation.



Zhi Zhou received the B.S., M.E., and Ph.D. degrees in 2012, 2014, and 2017, respectively, all from the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan, China. He is currently an associate professor in the School of Computer Science and Engineering at Sun Yat-sen University, Guangzhou, China. In 2016, he was a visiting scholar at University of Göttingen. He was nominated for the 2019 CCF Outstanding Doctoral Dissertation Award, the sole recipient of the 2018 ACM Wuhan & Hubei Computer Society Doctoral Dissertation Award, and a recipient of the Best Paper Award of IEEE UIC 2018. His research interests include edge computing, cloud computing, and distributed systems.

Fen Xiao received the master's degree from the School of Computer Science and Technology, Wuhan University, Wuhan, China, in 2009. She is currently a senior R&D engineer at Tencent, Shenzhen, China. Her research interests focus on NoSQL and distributed storage systems.

Fangming Liu (S'08, M'11, SM'16) received the B.Eng. degree from the Tsinghua University, Beijing, and the Ph.D. degree from the Hong Kong University of Science and Technology, Hong Kong. He is currently a Full Professor with the Huazhong University of Science and Technology, Wuhan, China. His research interests include cloud computing and edge computing, datacenter and green computing, SDN/NFV/5G and applied ML/AI. He received the National Natural Science Fund (NSFC) for Excellent Young Scholars, and the National Program Special Support for Top-Notch Young Professionals. He is a recipient of the Best Paper Award of IEEE/ACM IWQoS 2019, ACM e-Energy 2018 and IEEE GLOBECOM 2011, the First Class Prize of Natural Science of Ministry of Education in China, as well as the Second Class Prize of National Natural Science Award in China.