MasterPlan: A Reinforcement Learning Based Scheduler for Archive Storage

XINQI CHEN, Shanghai Jiao Tong University, Shanghai, China
ERCI XU, Shanghai Jiao Tong University, Shanghai, China
DENGYAO MO, Alibaba Cloud Computing, Bellevue, United States
RUIMING LU, Shanghai Jiao Tong University, Shanghai, China
HAONAN WU, Shanghai Jiao Tong University, Shanghai, China
DIAN DING, Shanghai Jiao Tong University, Shanghai, China
GUANGTAO XUE, Shanghai Jiao Tong University, Shanghai, China

With the sheer volume of data in today's world, archive storage systems play a significant role in persisting the cold data. Due to stringent cost concerns, one popular design is to organize disks into groups and periodically switch them to be powered on for serving user requests. Scheduling thus becomes critical for both CapEx and performance. Unfortunately, field results indicate that existing schedulers can be often suboptimal. Our further analysis suggests that the main reason is the mismatch between the ever-changing workloads and the fixed set of coarsely-configured parameters in current heuristic-based schedulers.

In this article, we propose `MasterPlan`, a reinforcement learning (RL) based scheduler for archive storage systems. By identifying the unique characteristics of archive storage service, we design a state space and reward function for the RL agent. `MasterPlan` includes a continuous action encoding approach to guarantee efficient exploration, and a meta adaptation module to extract features of workload series. Experiments show that `MasterPlan` can achieve 1.25× throughput, 2.16× 99^{th} latency and 1.47× power draw improvement compared to existing solutions.

CCS Concepts: • **Computing methodologies** → **Planning and scheduling**; • **Information systems** → **Storage management;**

Additional Key Words and Phrases: Archive storage system, reinforcement learning

ACM Reference Format:
Xinqi Chen, Erci Xu, Dengyao Mo, Ruiming Lu, Haonan Wu, Dian Ding, and Guangtao Xue. 2025. MasterPlan: A Reinforcement Learning Based Scheduler for Archive Storage. *ACM Trans. Arch. Code Optim.* 22, 1, Article 27 (March 2025), 25 pages. https://doi.org/10.1145/3708542

This work is supported by the Natural Science Foundation of China (62072306, 623B2072).
Authors' Contact Information: Xinqi Chen, Shanghai Jiao Tong University, Shanghai, China; e-mail: chenxq66@sjtu.edu.cn; Erci Xu (Corresponding author), Shanghai Jiao Tong University, Shanghai, China; e-mail: jostep90@gmail.com; Dengyao Mo, Alibaba Cloud Computing, Bellevue, Washington, United States; e-mail: dengyao.mo@alibaba-inc.com; Ruiming Lu, Shanghai Jiao Tong University, Shanghai, China; e-mail: lrm318@sjtu.edu.cn; Haonan Wu, Shanghai Jiao Tong University, Shanghai, China; email: haonanwu@sjtu.edu.cn; Dian Ding, Shanghai Jiao Tong University, Shanghai, China; e-mail: dingdian94@sjtu.edu.cn; Guangtao Xue, Shanghai Jiao Tong University, Shanghai, China; e-mail: gt_xue@sjtu.edu.cn.

ACM 1544-3973/2025/03-ART27
https://doi.org/10.1145/3708542

1 Introduction

In the cloud era, data is expanding at an unprecedented rate. However, more than 80% of the data (e.g., medical images and logs), while requiring permanent persistence, are infrequently or even never accessed after 90 days [6, 52].

To archive such an enormous amount of cold data, reducing CapEx is a priority. Both academia and industry have proposed various solutions for archive storage [2, 6, 8, 9, 11, 36, 47]. One popular design philosophy is to group the disks within a rack (i.e., a JBOD) by power supplies and cooling channels. During runtime, the system would rotate the disk groups and only power on a small proportion (one or two groups) for energy efficiency. An exemplary example of such a design is Pelican by Microsoft Azure [47].

Under this setup, the efficiency of the scheduler matters a lot. Inspired by similar practices in container node selection (e.g., K8s [7] and Borg [55]) and VM allocation (e.g., Protean [25] and CoSpot [29]), one typical approach is to develop a heuristic algorithm. By assigning weights to incoming archive and restore (i.e., write and read) requests, the scheduler calculates an overall score for each group and selects the highest-score group to spin up next.

However, traces from our widely deployed archive storage system in the field show that heuristic designs can fall short for the following reasons. First, real-world workloads can vary significantly over time and space, making one set of weights hardly fit for all. Second, restore (i.e., read) workloads, unlike previous assumptions, are nevertheless non-negligible in the field, further increasing the complexity of scheduling. This is because, unlike archive (i.e., write), restore requests cannot always be served immediately—they need to wait for the targeted group to be powered on before accessing. Third, there are also backend requests (e.g., data integrity checking and repairing) and different priority levels (i.e., Service Level Objectives, SLOs) that require careful orchestration. Although manually fine-tuning parameters of the scheduler can improve the throughput significantly, it is nonetheless not scalable and requires considerable effort given the large parameter space and the ever-changing workload patterns. In short, we expect the scheduler to be efficient, adaptive, and practical.

Reinforcement learning (**RL**) renders a fresh look on decision-making in a complex environment [37, 42, 43, 57] and multi-objective optimization [20, 34, 35]. The RL agent behaves as the central brain of the decision-maker and evolves in iterations. Our archive storage system possesses several properties (i.e., stable environment, no labeled data, and complex multi-objective control) that match the prerequisites for adopting an RL-based scheduling. Nevertheless, our initial attempts at migrating known RL models to our system failed. The main reasons are the large action space and the unreasonable action exploration.

Based on the above motivations and observations, we build an RL-based scheduler, named `MasterPlan`. The high-level idea is to train an RL agent (i.e., a policy neural network) to learn a fine-grained scheduling policy by maximizing a multi-objective reward. For each training episode, we feed system states to the policy network, which then generates an action for each disk group (e.g., spin-up/down). After actions are executed, `MasterPlan` collects updates of system states and calculates the rewards based on our goals (e.g., throughput and power draw). Rewards are then used to update the policy network and adjust the action selection. This process repeats until the policy network converges, indicating that the agent has learned the optimal scheduling policy.

The design of `MasterPlan` focuses on three aspects: First, we summarize the system properties of our archive storage service into a dedicated state space and design a reward function aggregating multiple objectives with assigned weights. The knack here is to identify the key metrics (e.g., throughput and power draw) that reflect the goals for the scheduler. Second, we design a continuous threshold-based action encoding method for each disk group. The rationale is that traditional practice (i.e., discrete action space [37, 41]) fails to appreciate the fact that disk groups

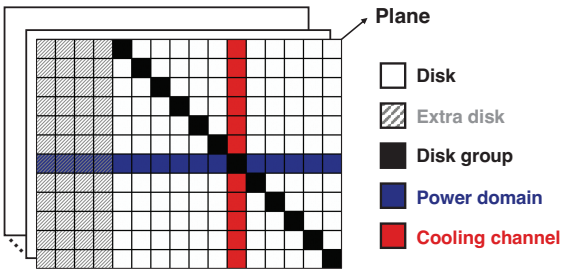


Fig. 1. Disk and domain layout of a Pelican [47] rack (Section 2.1). *Extra disks form disk groups with disks from other planes.*

should rarely switch between scheduling intervals to minimize spin-up latency and thus struggles to converge with an exponentially large action space $O(2^N)$ for N disk groups. Continuous action encoding outputs a constant value for each disk group, which not only reduces the action space for fast training but also enables us to apply system-wise limitations for accurate action generation. Third, we introduce a meta adaptation module to ensure the adaptability of the RL agent to different workload patterns. This module extracts temporal features from workload series and feeds them to the policy network, facilitating generalized training.

To evaluate the effectiveness of `MasterPlan`, we implement a high-fidelity simulator that mirrors our real-world archive system. We then conduct extensive experiments by comparing `MasterPlan` with other schedulers (e.g., **round-robin** (**RR**), K8s, and heuristic) and theoretical upper bounds (i.e., assuming no switching overhead, no power-on limitations, and unbounded network bandwidth). The results show that `MasterPlan` can achieve 1.25× throughput, 2.16× 99^{th} latency and 1.47× power draw improvement, and delivers performance (e.g., throughput) close to the theoretical upper bounds. Further evaluations also validate our design in each component and demonstrate that `MasterPlan` can adapt to various workloads and hardware configurations.

Here, we summarize our contributions as follows:

— We analyze field workloads and identify the root causes behind performance inefficiencies in existing archive storage schedulers.
— We propose `MasterPlan`, an RL-based scheduler for archive systems that can achieve fine-grained and adaptive control. Extensive evaluations show that `MasterPlan` can outperform existing schedulers and deliver close to upper-bound performance.
— We release `MasterPlan`, the archive system simulator, the production-driven traces, and the workload generator.[1]

The rest of this article is organized as follows. We introduce archive systems and scheduling details in Section 2. We then analyze the workload characteristics for motivation in Section 3, discuss a preliminary method in Section 4, the design and implementation of `MasterPlan` in Section 5 and Section 6. We evaluate `MasterPlan` in Section 7 and end this article with the related work in Section 9, and a short conclusion in Section 10.

2 Background

2.1 Archive Storage System

Disk-based archive storage systems are common in the field [1, 2, 47]. One notable example is Pelican [47]. Figure 1 describes the logical layout of a Pelican rack. Each rack is essentially a **JBOD**

[1] https://github.com/MasterPlanForSubmission/MasterPlan

(**Just a Bunch Of Disks**) system with two servers for management/scheduling, and thousands of disks (1,152 by default) for persistence.

For cost concerns, the disks in a Pelican rack are attached to different power supplies and cooling fans. Specifically, Pelican organizes disks as planes (see Figure 1). Disks on the same power supply (i.e., a row in Figure 1) can have at most two disks spinning simultaneously. Disks that share the same fan (the column) face a similar situation—allowing only one active disk per cooling channel. Thus, Pelican places disks into groups (the diagonal) that meet the constraints (i.e., disks in the same group are from different power supplies and cooling channels). In addition, Pelican structures each group as an **erasure coding** (**EC**) group for data redundancy and periodically switches them to serve requests.

Apparently, a standalone Pelican instance (i.e., a 52U rack) can not provide independent archival storage service to end users in the cloud. Here, we introduce the design of a complete archival system inspired by the Pelican prototype. The system is widely deployed (EB-level) in a major cloud vendor and has been serving more than thousands of users for many years.[2]

Figure 2 depicts a high-level organization of our archival service. The top layer is a distributed **Key-Value** (**KV**) service (e.g., HBase [3]) to let users store and access data using the common object store interface. Upon receiving the data, the archival system does not immediately invoke the archive storage clusters for persistence. Instead, the KV service will first flush data to a standard distributed file system (DFS, e.g., HDFS [48]) for temporal storage. Similarly, the requested data would be cached in the DFS after retrieval from the underlying archive pods. The benefits of this DFS-based cache are two-fold. First, end users can read/write data just like using a standard cloud object storage service (except reading data requires an unfreezing waiting period). Second, the DFS-based cache can merge small requests into large files (e.g., 10 GB) and then write to underlying archive pods. The bottom layer is the archive storage cluster that consists of multiple archive pods. Each pod includes at least one server to schedule I/Os and switch the disk groups. Additionally, each pod includes one or more Pelican-like racks and organize the disks as Figure 1 shows.

Archive systems usually have a rather relaxed latency requirement (e.g., hours or even days, called data unfreezing time). Therefore, disk-based archive storage systems, like Pelican, can reduce power consumption by exploiting this unfreezing time to switch disk groups to be powered on via a scheduler. Hence, the efficiency of the scheduler is critical. If the scheduler is suboptimal, the system would spend considerable time on switching and warming up devices (i.e., spin up/down disks) instead of serving requests.

2.2 Archive Storage Scheduling

We now describe the scheduling workflow of our archive storage system. There are typically four types of requests to serve, namely *archive*, *restore*, *scrub*, and *repair*. Archive and restore requests are write and read requests sent by users, respectively. In addition to user requests, our service occasionally schedules backend system requests like data checking (called *scrub*), and EC reconstruction (called *repair*) to ensure data integrity.

Figure 3 describes the scheduling workflow. There are generally three steps: ❶ Frontend user requests (i.e., archive and restore) and backend system requests (i.e., scrub and repair) are enqueued to the corresponding **disk group queue** (**DGQ**).

❷ Requests are dispatched by the I/O scheduler to different types of queues. Note that users can choose one of the following three SLO levels for restore requests: low (<12 hrs), high (<4 hrs), and urgent (<1 hr). Archive and backend requests are set to low SLOs by default. The I/O

[2]For anonymity, we do not disclose the name of our system and use similar open source software as examples for illustration.

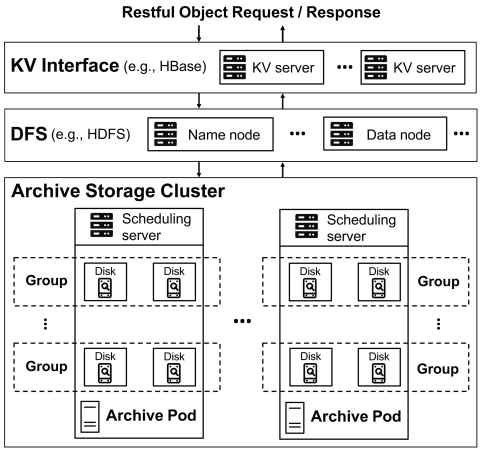


Fig. 2. Overview of our archive storage system architecture (Section 2.1).

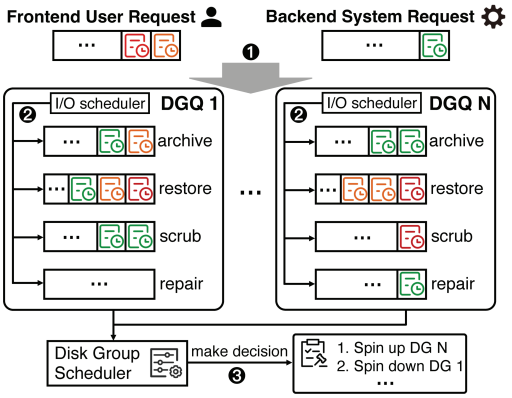


Fig. 3. Scheduling workflow of the archive system scheduler (Section 2.2). ***DG:*** *disk group;* ***DGQ:*** *disk group queue. Different colors indicate the priority of requests.*

scheduler[3] follows a priority-based mClock algorithm [24], inserting requests into the designated queue according to their SLOs, and it periodically reorders for changes (i.e., low SLOs become higher due to waiting).

❸ The disk group scheduler selects candidates to spin up/down by calculating the overall weights of requests queued in each disk group. Specifically, each request is assigned a score based on its SLO and size. The scheduler then selects the disk group with the highest accumulated scores as the next candidate to spin up, and the group with the lowest scores to spin down. Like Pelican, at most two disk groups can be powered on due to power/cooling constraints. Note that the disk group scheduler also adds an extra score to the currently spinning group, places a hard limit on the minimum/maximum serving time, and enforces an interval between two spin-ups of the same disk group to avoid frequent and unnecessary switching.

Other archive systems have also implemented similar heuristic-based solutions. For example, Pelican maintains a global queue for all the read requests, and the disk group scheduler reorders the requests by batching them based on the target disk groups. Consider that the scheduler has queued four requests targeting four different disk groups, say [A, B, C, and D] from first to last. Now, for a new request to disk group A, the scheduler would reorder the queue as [A, A, B, C, and D]. As another example, Silica [9] simply selects the glass platter in a FIFO manner and plans multiple robot movement paths using the A^* algorithm [13].

3 Motivation

3.1 Studying Field Traces

We have collected traces from 9 production archive storage clusters of four different available zones, where each cluster has several Pelican-like racks and holds PB-level storage capacity. We observe that the archival workloads can contradict the common knowledge in the literature, as they show significant temporal and spatial variations, a non-negligible proportion of restore requests, and multiple priority levels along with their impacts on user experiences.

[3]The performance impact of the I/O scheduler is not as important as the disk group scheduler. Therefore, we focus on studying the disk group scheduler for the rest of the article.

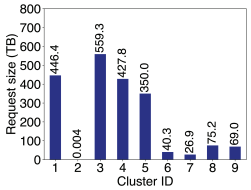


Fig. 4. Request sizes of 9 different production clusters over a single day (Section 3.1).

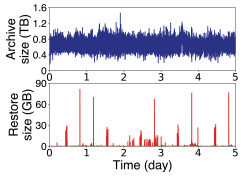


Fig. 5. Archive and restore request sizes of a production cluster over 5 days (Section 3.1).

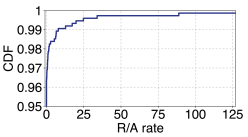


Fig. 6. CDF of r/a request size ratio in a five-min window over a month (Section 3.1).

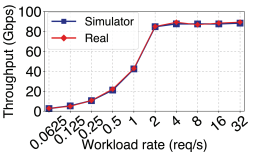


Fig. 7. Cross validation results of system throughput (Section 3.3).

Dynamic workloads. Archive storage systems can have workloads with rather varying access patterns. Figure 4 displays the total size of requests in 9 archive clusters over a single day, revealing notable imbalances. Even within the same cluster, the requests—both restore and archive—also vary significantly over time as shown by Figure 5. This contradicts the previous assumptions that archive system workloads are mainly batches of stable incoming write requests [36, 47], which can be easily dispatched using a (manually tuned) heuristic scheduler. For highly dynamic workloads, the coarse-grained nature (i.e., assigning each type of operation with a fixed score) of the heuristic scheduler across all clusters can lead to subpar performance.

Non-negligible restore. Another popular opinion is that the archive storage system, especially the scheduler design, should mostly focus on write (i.e., archive) requests as read (i.e., restore) requests for cold data can be fairly infrequent [1, 36]. However, our observation indicates that restore requests can be non-negligible in practice. As shown in Figure 6, we compute the restore and archive request size ratio in five-minute windows of a production cluster over a month. Although the archive size is larger than restore in 97% of the time, we notice that restore can be even 25× larger during bursts. Additionally, in field deployment, local backend scrub traffic can be 5× higher than archive traffic to ensure robust data integrity, which thus affects the scheduling decisions. This means that the scheduler should take the varying task types into consideration (e.g., for performance isolation) instead of solely focusing on archive workloads.

Multiple priority levels. Previous archive systems [38, 47] assumed that all requests have equivalent urgency levels. However, real-world service usually provides users with varying SLO priorities, such as 10 mins, 5 hrs, and 10 hrs level in AWS [2], 1 hr and 15 hrs level in Azure [4]. Due to the high costs of urgent operations, most restore requests are assigned a low priority by users. However, from our field observations, there are still 10% of medium and high priority requests to meet pressing needs. Besides, backend requests also have corresponding priorities. For example, a scrub request following a longer interval since the last integration will have a higher priority. Further, while various requests are assigned priorities upon creation, the priority of unaddressed requests escalates over time. The scheduler should embrace this ever-changing nature of queued requests and adapt accordingly by placing higher priority on urgent tasks.

3.2 Existing Scheduler Considered Inefficient

In field deployment, while the archive system successfully serves the cold data storage needs of various users, we observe that the clusters do not deliver consistent performance. For example, Figure 8 plots the throughput variations of two racks during 1-hour monitoring. With a lighter workload pressure (i.e., restore+archive size of 41.8 TB+25.2 TB vs. 55.6 TB+38.6 TB), the left rack delivers a higher throughput (i.e., 96.1 Gbps vs. 89.2 Gbps). Upon visual inspection, we can also see

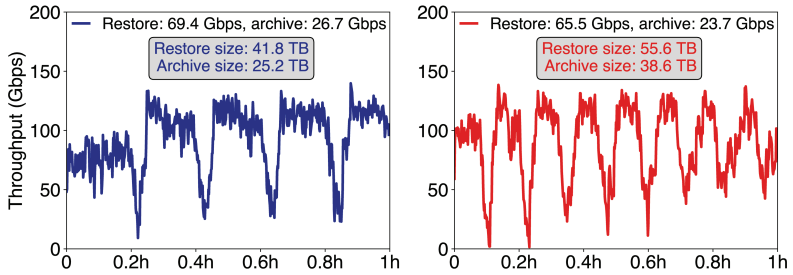


Fig. 8. Rack throughput in one hour under two different workloads (Section 3.2). *The lighter workload (left figure) achieves a higher total throughput performance (96.1 Gbps vs. 89.2 Gbps).*

that the right rack has conducted more disk group switching operations (i.e., throughput spikes), thereby spending more time on spin-ups. This suggests that the existing scheduler could be improved to better adapt to the workload variations and achieve higher performance.

We identify that the subpar performance occurs because the archive system uses the same set of disk group scheduler parameters, such as the request score weight and serving time threshold for all scenarios, which is unlikely to be optimal, and consequently leads to performance differences. For instance, the parameters can favour high SLO requests for their urgency, but the currently spinning disk groups can also be prematurely replaced due to these urgent requests.

3.3 Archival Simulator

To validate our assumptions and explore alternative solutions, we build a simulator that accepts request-level traces to quantitatively analyze scheduler behaviors under various scenarios. The simulator provides interfaces for archive systems with other storage media (e.g., tape) to facilitate comprehensive evaluations. The reason for using a simulator instead of directly testing on production systems is two-fold.

First, configuring the archive systems can be prohibitively time-consuming. For example, to explore the performance impacts of disk group formation, we might even need to reorganize power domains and cooling channels completely. Second, evaluating archive storage systems via simulation is a common practice (e.g., Pelican [47]) and can achieve high precision. This is because the archive system operates at a coarse granularity—switching disk groups at a long interval (i.e., minutes). In addition, archive systems serve relatively less intense workloads (at most several requests per second during deployment), thus usually not bounded by hardware resources like CPU and memory. Therefore, the stack is unlikely to be influenced by usual variations that prevent high-precision storage stack simulation (e.g., µs-level I/O or page delay [33]). From Figure 7, it can be seen that our simulator can accurately mimic the behaviors of the production system from low to high I/O intensity (i.e., 0.0625 to 32 requests per second). See Section 7.1 for more details on the simulator.

3.4 Results

We now evaluate the performance with three production-driven traces (Trace 1, 2, and 3 in Table 1) and two simulated ones (Simulate 1, 2). The simulated traces place a higher intensity (16 req/s and 32 req/s, respectively) to reflect the pressure under bursty workloads with batch restore and constant archive requests. Note that bursty workloads indeed happen during real-world production (e.g., users archive data incorrectly and restore them at once, excessive archive/restore behaviors from large users, etc.), and similar phenomenon can be found in [9]. Apart from the disk group

Table 1. Throughput (Gbps) Results using Different Schedulers (Section 3.4)

#	Trace 1	Trace 2	Trace 3	Simulate 1	Simulate 2
DS	71.51	72.44	72.28	/	/
SS	71.33	72.52	72.34	88.12	90.37
P1	**74.65**	73.24	71.98	85.97	86.06
P2	73.23	**78.91**	77.37	**101.66**	97.58
P3	70.33	69.37	**79.59**	97.19	**102.27**
KS	70.36	67.21	65.21	85.21	85.98
RR	69.62	65.35	61.50	84.97	85.09
Rd	58.57	58.30	42.47	64.78	66.33
DR	52.12	53.24	49.15	56.12	60.58
MP	76.24	79.43	81.94	103.57	104.61

DS: *deployed system;* ***SS:*** *simulated heuristic scheduler;* ***Pi:*** *heuristic scheduler that tunes parameters specifically for trace i;* ***KS****: K8s scheduler;* ***RR:*** *round-robin;* ***Rd:*** *random;* ***DR:*** *Decima RL agent;* ***MP:*** `MasterPlan`*. "/" indicates that it cannot be tested.*

schedulers, other system parameters (e.g., using priority-based I/O scheduler and number of disk groups) are the same as deployment setups.

The first row of Table 1 is the actual throughput of the field deployment, thus simulated trace results are not included. The second row is the throughput of the simulator under our heuristic scheduler with default parameters (same as the deployed system). The next three rows (P1 to P3) are the results by fine-tuning the minimum scheduling time interval and request score of heuristic schedulers for corresponding traces (Trace 1 to 3). The KS row is the throughput derived from the K8s scheduler. The RR and Rd rows report throughput when both switched at the same fixed interval (10 minutes) but choose disk groups in a RR or random fashion, respectively. We modify the Decima RL agent (DR) to adapt to our system, and we will discuss the design of DR and `MasterPlan` (MP) later in Section 5.

We make the following observations based on Table 1. First, the simulation results are close to the actual deployment (within 0.25% difference) as expected. In addition, we can see that the choice of disk group switching does matter, as RR and Rd constantly deliver subpar performance. Moreover, fine-tuning clearly helps the rack to achieve higher throughput (see the bold numbers from P1 to P3 in Table 1). This verifies our hypothesis that the existing "one set of parameters fits all" switching strategy is not optimal. In addition, by comparing the throughput across all five traces, we can see that the ranking of P1 to P3 is inconsistent, indicating that the tailored tuning may only capture certain but not general workload characteristics.

4 Exploring a Better Design

4.1 Why Not Fine-tune All Clusters?

Intuitively, the preliminary results in Table 1 indicate that we can manually configure the disk group scheduler for better performance. Unfortunately, this practice is not scalable and subject to workload changes over time. Previous results in Section 3.4 have indicated that tailored tuning cannot provide consistent performance across different workloads. Therefore, one needs to constantly and individually fine-tune each cluster, which is impractical given the massive number of machines. Furthermore, the scheduler also needs to simultaneously optimize many competing objectives (e.g., maximize throughput, minimize power draw), and thus further involves dozens of hyperparameters (e.g., score weights), yielding a massive parameter space.

Despite the time overhead and manual effort, a more profound reason preventing us from focusing on improving the heuristic scheduler is its coarse-grained nature. Under this strategy, the same score for different operations (e.g., restore and scrub) applies to all disk groups. Ideally, even for the same operation, we should be able to assign different scores to each disk group and periodically update them to better adapt to workload variations. However, under the current design, this is not possible as it adds yet another layer of complexity to the already complicated heuristic scheduler tuning.

At this point, we can summarize the characteristics of an ideal scheduler in archive storage systems as follows. First, a sophisticated scheduler should be *fine-grained* for precise system control. Second, the scheduler should automatically learn from the environment and thus be *adaptive* to various workload patterns. Third, an ideal scheduler is supposed to achieve *near-optimal* performance while balancing multiple objectives with acceptable resource consumption.

4.2 Why Explore RL-based Scheduling?

In the search for a solution, RL has caught our attention for its effectiveness in similar scenarios. Previous works have successfully solved various scheduling problems using RL, such as job scheduling (Decima [37] and SchedInspector [57]), workload autoscaling (AWARE [43]), and congestion control (AUTO [34] and MOCC [35]). We discover that the setup of an archive storage system bears several similarities with these works, thereby yielding potential opportunities for applying RL in our case.

Advantage 1: Stable Environment. AWARE points out that altering the underlying infrastructure can lead to a time-consuming overhaul on the RL policy retraining. Fortunately, similar to Decima and AUTO, the setups of archive storage system, including disk group formation and EC configurations, are rather stable. This is because that in a switching-based archive system, the formation of a disk group depends on the physical layout of power supplies and cooling fans, and thus rarely alters after installation. In addition, the choice of EC configuration (i.e., the number of data and parity disks) is usually determined before deployment and remains unchanged during operation to keep the data persisted in a consistent format.

Advantage 2: Unlabeled Dataset. Many traditional ML techniques [19, 54] require abundant labeled data for training, which is difficult for us to collect and categorize (e.g., determining and labeling a spin up action as positive or negative). However, RL only requires a reward function to guide the learning of the optimal policy. The agent can explore the action space without the need for labeled data as supervision, thus mitigating the high sample complexity problem.

Advantage 3: Complex Optimization. We expect our scheduler to optimize multiple objectives, such as throughput, power draw, and SLO, which are analogous to AUTO and AWARE. Especially, we aim to optimize the long-term goals, such as the overall throughput instead of an instantaneous one. In this scenario, RL can conveniently adopt a delayed and weighted reward to achieve the above goals.

5 `MasterPlan` Design

5.1 System Overview

Motivated by the above advantages, we develop `MasterPlan` as shown in Figure 9. The high-level idea is to build and train an RL agent (i.e., the policy network) to learn the optimal decisions for disk group scheduling.

For training, `MasterPlan` first models the system states (i.e., disk group and request states, ❶) from the simulated environment as an N-dimension vector using a state collector, where N is the number of disk groups. `MasterPlan` then feeds the vector into the policy network, which outputs

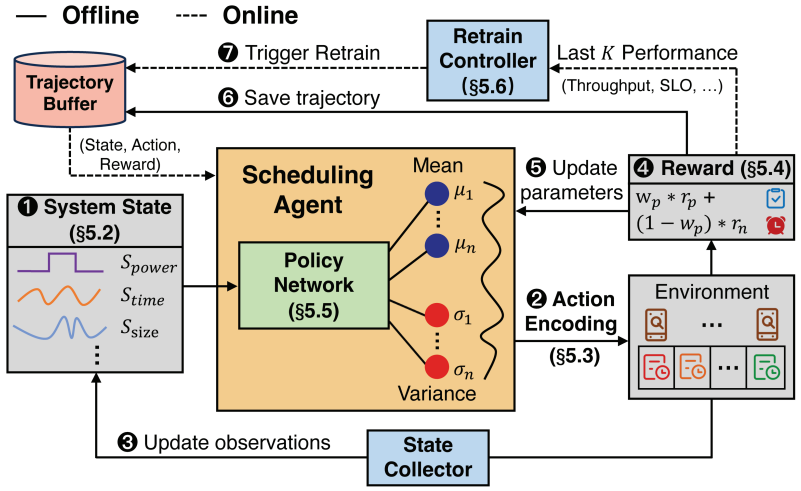


Fig. 9. Basic framework of MasterPlan (Section 5.1).

an encoded action (e.g., spin up, ❷) for each disk group. Afterwards, archive system behaviors are simulated under the output actions (❸) for a fixed interval (five seconds by default), and a reward is calculated based on the current system states. The reward calculation (❹) is a weighted sum of our objectives, including throughput, power draw, and so on. Finally, MasterPlan updates the system states and repeats the above process. After a batch (256 by default) of such explorations, trajectories (i.e., time series of state, action, and reward) are fed back to the policy network (with a meta adaptation module described in Section 5.5) for parameter updates (❺), which are then saved in the trajectory buffer (❻). This iterative process continues until the model converges.

After we train a converged agent (i.e., the policy network), we deploy it online. Similar to training, we feed the updated system states at a fixed scheduling interval, and MasterPlan selects an action based on the policy network output. When MasterPlan observes subpar performance in a predefined time window, MasterPlan would retrain the policy network using buffered trajectories (❼). For the rest of this section, we discuss the design of *system state* (Section 5.2), *action space* (Section 5.3), *reward function* (Section 5.4), *meta adaptation* (Section 5.5), and *retrain strategy* (Section 5.6) in detail.

5.2 System State Design

Designing an effective state space is critical, as not capturing the key metrics would result in the RL agent failing to make optimal decisions to meet our goals. In MasterPlan, we include four types of states (17 in total) as the state vector for each disk group, including disk group power state, disk group serving time, request size, and urgency of four request types. This enables the collection and derivation of important statistics including overall throughput, SLO violations of each request, and total power draw. For the robustness and effectiveness of model training, we employ state normalization and calculate the relative value for each feature. For a given disk group i, we now describe the state features as follows. Note that by aggregating these state features, an $N \times 17$ matrix can be formed to feed the policy network.

1. Disk group power state $S^i_{power} \in \mathbb{R}^4$**.** We use four dimension *one-hot* encoding (i.e., 0001, 0010, 0100, and 1000) to represent the power state of each disk group, including serving, idle, spinning up and spinning down. Note that spinning up and down can cause different time-/power-consuming, so we treat these two as independent states.

2. Disk group serving time $S^i_{time} \in \mathbb{R}^1$. We record the serving time of a disk group as an integer. The time is zero if the group is currently idle or spinning up/down.
3. Request size of different types $S^i_{size} \in \mathbb{R}^6$. For each disk group, there are six priority-based I/O queues (three for restore requests corresponding to three different SLOs), and each contains one type of request, including archive, restore, scrub, and repair. Here, we use six integers to represent the total task size of each queue to reflect the overall workload pressure.
4. Request urgency of different types $S^i_{urgent} \in \mathbb{R}^6$. We represent the waiting time of the first request in each queue with six integers. In other words, for the archive, restore, repair, and scrub queue, we check the first in line of each queue and calculate the waiting time by deducting the request's arrival time from the current time. Initially, we have explored other representations, such as the average request waiting time of each queue. However, the urgency of certain requests may be buried in the metrics, yielding frequent SLO violations. Here, we utilize the inherent attributes of priority queues as requests are constantly (re)ordered based on sizes and the remaining time to SLO violations, thus the head request characterizes the urgency of its queue.

5.3 Encoding Action Space

Discrete action RL has been widely implemented in previous works (e.g., congestion control [30, 53, 56]), and the control mode of disk groups (i.e., spin up/down or stay unchanged) is quite suitable to this discrete encoding feature. However, directly applying this design to archive storage scheduling faces significant challenges and yields subpar performance. Now, we first introduce a discrete action space design and a straightforward pruning technique proposed in Decima before we present our *continuous action encoding* method.

5.3.1 Naive Action Space Design. Similar to the disk group power state design, one intuitive approach to representing the action space is through *one-hot* encoding. In this case, we map all possible actions of each disk group to a $1 \times V$ vector, where V is the number of possible actions. For example, if we have N disk groups with up to n spinning, the total action space will grow from $O(N^2)$ to $O(2^N)$ when n increases from 2 to N. This is because the number of spin up actions is formulated as $\sum_{i=1}^{n} C_N^i$, where C refers to the combination notation. Having a large space can be problematic for RL training, as it is difficult for action exploration and model convergence.

Motivated by solutions in Decima, we attempt an alternative method by decomposing the huge action space into *multi-discrete* actions. Specifically, we generate a discrete action for each disk group and then concatenate them into an action matrix. Since we have N disk groups and each group can take 3 actions (i.e., spin-up/down or stay unchanged), the encoded action is an $N \times 3$ matrix.

However, even if we reduce the action space size to $O(N)$, the RL agent still struggles to converge. There are two primary inefficiencies. First, the agent starts by selecting actions randomly. Hence, especially during the initial exploration, the agent would constantly switch between disk groups as the probability of staying unchanged is only one-third. Therefore, the agent would consistently receive low rewards and struggle to converge since the policy network fails to appreciate the long-term benefits of staying unchanged.

Second, recall that only a small number of disk groups are permitted to power on simultaneously. Therefore, when the policy network outputs an action to command more-than-allowed disk groups to spin up, `MasterPlan` must decide which groups to spin up and which to ignore. However, with multi-discrete action encoding, the actions on different disk groups are equivalent. For example, for disk groups A, B, C, and D, choosing A and B to spin up is the same as choosing C and D from the encoding's perspective. The two selections would behave differently due to varying request urgency in each DGQ, contributing to the slow convergence.

5.3.2 Continuous Action Encoding. Previous attempts (one-hot and multi-discrete encoding) have demonstrated the need for a smaller action space and better control of action selection. Ideally, scheduling for disk groups should choose "stay unchanged" actions for most intervals, with a small fraction of spin up/down actions. In order for the agent to capture this action pattern during early training, behavior cloning has been used in other works [27, 39, 49] for its guidance through supervised learning using expert data. Nevertheless, imitating the heuristic experience can lead to noise and misidentification [18, 50]. To address these, we decide to enable the agent to self-learn by attempting *continuous action encoding*.

Specifically, the actor network outputs N-dimensional actions, each representing an action value a_v for the corresponding disk group. By constraining a_v within the range of -1 to 1, we can employ a threshold T to distinguish different actions:

$$Action = \begin{cases} Spin\ up, & a_v > T \\ Stay\ unchanged, & -T \le a_v \le T \\ Spin\ down, & a_v < -T \end{cases} \tag{1}$$

Note that applying a spin up/down action to a disk group in the same state is equivalent to staying unchanged.

This encoding method offers three advantages: First, it condenses the action space from one-hot actions to a single action per disk group, thereby reducing the space size to $O(N)$. Second, by adjusting T in continuous encoding, the likelihood of selecting the stay unchanged action can be set to a suitable probability (e.g., 80% for $T = 0.8$). In this way, the frequent switching phenomenon of disk groups can be mitigated, which greatly enhances training efficiency. Third, since actions of each disk group are encoded to a value a_v, the agent can prioritize disk groups with a larger a_v to spin up. It encourages the agent to learn the importance of spinning up actions among different disk groups, and better integrates with the hardware constraints of archive systems.

5.4 Reward Function

Given the system state S, our optimization objectives are to maximize throughput ($g_1(s)$) and adherence to SLOs ($g_2(s)$), minimizing power draw ($g_3(s)$) and request latency ($g_4(s)$), formulated as: $max(g_1(s), g_2(s), -g_3(s), -g_4(s))$, where the minus indicates that it is a penalty term. Unlike traditional RL which aims to optimize a single scalar reward, a multi-objective problem aims to find the set of Pareto-optimal solutions, which depends on the relative preferences among competing criteria. Training RL agents with a final episode reward proves challenging due to the sparsity of environment feedback [10]. Hence, we adopt the reward shaping method [23] which generates more frequent rewards to guide the agent training. In this case, the initial multi-objective problem can be mixed into a single reward through linear weighted combination [42].

We define positive and negative rewards to describe the action effects on the final goal. For positive reward, we use rewards of throughput (r_p^{tp}), and finished requests (r_p^{fin}) between each interval to incentivize the agent to achieve high throughput and ensure request SLOs. For penalty, we define rewards of request latency (r_n^{lat}), serving disk group number (r_n^{sn}), and disk group serving time (r_n^{st}) that penalize the agent for minimizing latency and power draw. Besides, if the agent spins up (r_n^{up}) or spins down (r_n^{dw}) disk groups in advance (less than five minutes), it will also incur a substantial penalty. We employ the reward scaling [26] method to control rewards within an appropriate range of (-1, 1). To sum up, the total reward function is formulated as: $reward = w_p * r_p + (1 - w_p) * r_n$, where w_p is the weight ratio of positive rewards, r_p and r_n are the above rewards by linear accumulation.

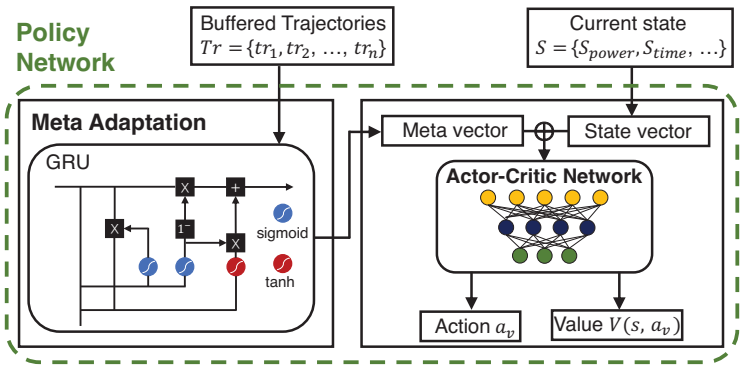


Fig. 10. Breakdown of the policy network (Section 5.5).

5.5 Meta Adaptation

Traditional RL agents require extensive training samples for outstanding performance. However, as discussed in Section 3.1, our user workloads are dynamic and varying through time. Therefore, the performance of RL agents may diminish in practice when encountering previously unseen workload patterns. Although retraining could help, it is time-consuming to retrain the agent, not to mention the workload patterns could change again during retraining.

To address this challenge, we leverage meta learning [21, 22] to help the RL agent comprehend the more general and underlying patterns, rather than overfitting to specific data. In a nutshell, meta learning allows the agent to "learn how to learn". In this way, `MasterPlan` can learn the temporal characteristics of incoming workloads, while retaining the capability to infer from previously encountered data.

As shown in Figure 10, we use a **gated recurrent unit** (**GRU**) neural network [51] as the feature extractor to encode the continuous coming time-series data. GRU employs a memory gating mechanism to preserve temporal semantics. Compared to **long short-term memory** (**LSTM**) networks, GRU is more efficient in capturing features while maintaining equivalent performance. In the policy network of `MasterPlan`, buffered trajectories (state, action, reward pairs) are encoded into a meta vector that includes information on different workloads. The resulting vector is concatenated with the state vector as the input, and model parameters are jointly updated. Under this design, the agent can learn to adapt to various workloads by observing both the meta vector (time dimension) and the state vector (space dimension).

5.6 Retrain Strategy

After an agent is trained and deployed, workload variances may still lead to performance degradation when encountering previously unseen patterns. To mitigate this issue, `MasterPlan` leverages a *retrain controller* to monitor the real-time performance. `MasterPlan` executes performance checking after a fixed time interval (e.g., 2 minutes). If the last K performance metrics (e.g., throughput) and reward fall below their corresponding performance thresholds, the controller triggers a retraining process with every workload trace that causes subpar performance, by exploiting the (state, action, and reward) pairs. `MasterPlan` can automatically fine-tune the policy network with collected trajectories, in contrast to heuristics schedulers that require extensive parameter-tuning efforts.

6 Implementation

We utilize the **Proximal Policy Optimization** (**PPO**) training method [46], which is eligible for continuous action encoding. Compared to offline RL algorithms, PPO enables the agent to adapt to

ALGORITHM 1: Training Process of `MasterPlan`.

Input: Number of epochs N_e, update frequency U_f, scheduling interval t_i.
Output: Trained parameters θ_a and θ_c for actor and critic network.

```
Initialize actor and critic networks with weights θ_o
for episode=0,1, ...,N_e do
    l_e ← f_t(episode)
    // Reset the simulator and get the initial state
    s_t ← RESET()
    d_t ← False, t ← 0
    while !d_t do
        log_p ← π(a|s; θ_a)
        a_t ← log_p.sample()
        // Step the environment with the sampled action
        s_{t+1}, r_t, d_t ← STEP(s_t, a_t, t, l_e)
        t ← t + t_i
        Buffer.append(s_t, a_t, s_{t+1}, r_t, d_t)
        if !d_t then
            s_t ← s_{t+1}
    if Buffer.size() ≥ U_f then
        dθ_a ← ComputeALoss(Buffer)
        dθ_c ← ComputeCLoss(Buffer)
        Update θ_a and θ_c using dθ_a and dθ_c
        Buffer.clear()
return θ_a, θ_c
```

real-time updates (i.e., changes in system states), and optimizes the summation of reward r_i with a decay factor γ: $Reward = r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots$, which prefers long-term cumulative rewards instead of instantaneous ones.

Moreover, we use the Actor-Critic [31] policy to enhance the model's robustness and accelerate training [40, 44, 45], as shown in Figure 10. It utilizes an actor network and a critic network with the same architecture (a three-layer fully connected network). The actor network outputs scheduling actions, while the critic network estimates the expected cumulative rewards of the state, providing a critic value of the current action. Besides, we employ a curriculum learning [12] strategy to prevent the agent from receiving massive negative rewards in the early stages. Unlike traditional RL that learns from a single task (i.e., fixed training traces), curriculum learning devises a curriculum that gradually increases the difficulty of tasks. Specifically, we implement an early-stopping technique to ensure stable training, depicted as:

$$f_t(idx) = t_0 + \lfloor idx/e_u \rfloor * t_i, \tag{2}$$

where t_0 is the initial training time step, idx is the current training episode index, e_u is the update episode interval, and t_i is the update time step length. It gradually expands the max training time step based on the episode iterations. Typically, for a 6-hour production trace, the training converges around 0.5–1 hour running on a 3.2 GHz CPU. Besides, the retraining is incremental and only takes minutes.

Algorithm 1 outlines the pseudocode for the entire training process. The training progresses with a fixed scheduling interval, while the episode length l_e gradually increases according to

Table 2. Three Possible Parameter Sets for an Archive System (Section 7.1)

System parameters	1	2	3
EC group	(20 and 4)	(18 and 3)	(8 and 3)
NIC bandwidth (Gbps)	200	400	800
Number of racks	2	4	4
Number of disk groups	36	60	48
Number of disks in a group	48	48	64
Number of spinning disk groups	8	16	12

We use the first set for deep evaluation.

Equation (2) (line 3). When the global time step reaches the schedule slot, various system contexts (e.g., task size and latency) are captured as state features (line 5). Subsequently, these features are fed into the policy network to generate an action, which is then applied to the environment for the next state, and the reward is calculated and provided as feedback for network updates (lines 7–15). During each loss computation step (lines 16–20), the buffered data is reused multiple times to enhance the training efficiency.

7 Evaluation

7.1 Experiment Environment

Simulation setup. To thoroughly evaluate `MasterPlan`, we use our simulator to emulate various setups. Table 2 outlines three representative setups, covering different numbers of disks per rack and varying EC scheme configurations. Based on our deployment experience, we set the disk read (mostly random) and write (all sequential) bandwidth to 50 MB/s and 100 MB/s, respectively. Similarly, the spin-up and spin-down overheads are set to be 60 seconds and 30 seconds. We collect synthesized traces to evaluate `MasterPlan` under various levels of I/O pressure. We use a Poisson distribution to generate the arrival time of requests, and the average arrival rate λ is set from 0.0625 to 32, indicating the number of requests per second. Additionally, we collect two field traces to demonstrate real scenarios of bursty (1) restore requests, and (2) backend requests. Finally, we set the disk power draw as: P_{spinup} = 24.1 W, P_{active} = 10.3 W, and $P_{standby}$ = 1.7 W. Our field statistics indicate that the power draw difference between reality and simulation is fairly small (less than 3%) when measured on an hourly basis. Note that when the disk group is spinning up, all its disks are in the state of spinning up. Additionally, when the disk group is serving, its disks can either be active or standby. During a disk group's idle or spin-down state, all its disks consume no power.

Test candidates. We compare `MasterPlan` (MP for short) with other disk group schedulers, including random, RR, Pelican's reorder, K8s and Silica's default scheduler and the deployed heuristic scheduler. Moreover, we simulate three **Full Provision (FP)** setups [47] to show the gap between MP and theoretical upper bounds. An FP-nospin setup uses the heuristic scheduler and assumes that switching groups does not incur spin-up/down overhead, but still only a limited number (2 by default) of groups can serve I/Os. FP-network assumes that all disk groups can serve I/Os, but the network bandwidth is limited to 200 Gbps. FP-unbounded assumes all disk groups can serve I/Os and the network bandwidth is unlimited.

7.2 Metrics

Throughput. We measure it by calculating the total amount of transferred data during the experiment time, demonstrating the request processing ability of the system.
Latency. Latency measures the time interval between the request issuance (i.e., the start of the serving not the completion) and initiation.

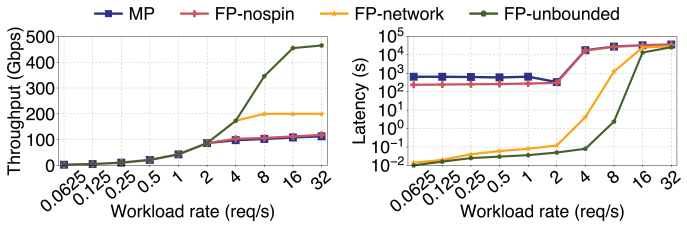


Fig. 11. Base performance comparing with FPs (Section 7.3).

Power draw. It measures the average power consumption of HDDs in the system. We do not include other hardware, such as the power of cooling fans, CPU, and memory.
SLO violation rate. This is the proportion of user's requests that violate the configured SLO requirements.

7.3 Comparing with FPs

First, we assess the system throughput under different workload intensities with a 0.2 r/w request ratio that reflects a real scenario. Figure 11 shows the throughput and latency performance of MP compared with FPs. As for throughput, we can observe that all candidates show similar performance between 0.0625 to 2 req/s, indicating MP can achieve top-level performance under light workloads. Although MP exhibits increased latency due to fewer spinning disk groups, it saves power while ensuring the SLOs. Even under high-intensity workloads (i.e., 4 req/s onwards), MP still achieves 94.0% of the FP-nospin throughput on average. This verifies that the scheduling efficiency of MP since the switching only incurs a minor throughput penalty. Similarly, the latency of MP is comparable to FP-nospin. The latency gap between MP, FP-network, and FP-unbounded is understandable as these two FPs do not wait for switching to serve I/Os.

Power draw evaluations are also conducted, as shown in Figure 12. The power draw remains constant at 17.8 kW for an always active FP-unbounded. In stark contrast, the peak power of MP is only 3.63 kW, accounting for merely 20.0% of FP-unbounded. Furthermore, MP reduces power draw by 46.5% compared to other schedulers at low workload rates due to its dynamic adaptation ability. When the workload rate exceeds 2 req/s, the power of MP approaches to other schedulers. For a 10-pod archive storage cluster, MP can save 2,000 $ CapEx compared to other scheduler, and 16,000 $ CapEx compared to FP-unbounded per month. Note that we only consider the CapEx savings from disk power draw.

7.4 Comparing with Other Schedulers

In Table 1, we present preliminary evaluation results of throughput. MP outperforms all other peers including the fine-tuned heuristic schedulers (P1 to P3). We evaluate the performance of various disk group schedulers under different intensities in Figure 14(a). Random exhibits poor performance on all metrics as it disregards the request characteristics and serving time of each disk group, leading to latency fluctuations due to its random nature. RR outperforms the random one since it serves all disk groups equally. Pelican's reorder scheduler aggregates restore requests to spin up disks, which results in longer latency under heavy workloads. The heuristic scheduler encounters challenges when many urgent requests are queued in each disk group, leading to frequent power switching and significantly affects performance. Although Silica exhibits high throughput, this results from a disregard for request initiation time, which is unacceptable given its numerous SLO violations. MP leverages RL to learn in different workload scenarios and achieves 1.25× throughput, 1.63× latency and 2.16× 99^{th} latency improvement compared to the heuristic

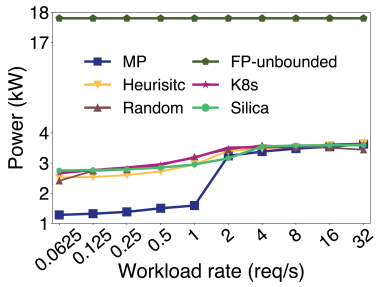


Fig. 12. Power draw of different disk group schedulers and FP-unbounded (Section 7.3).

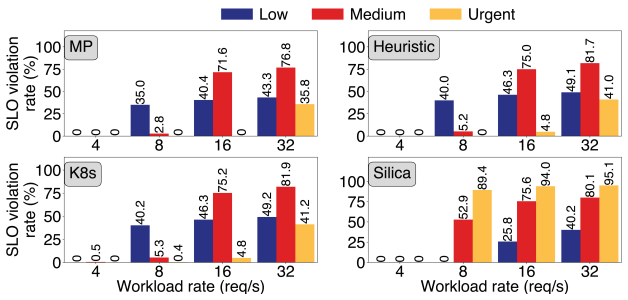
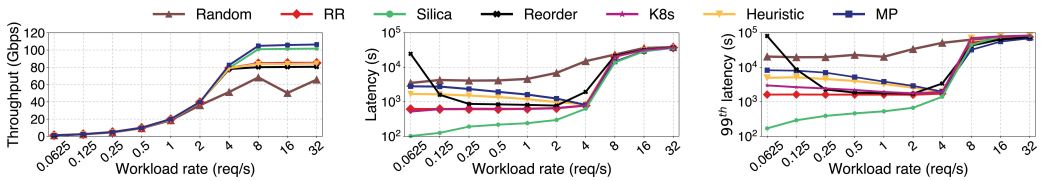
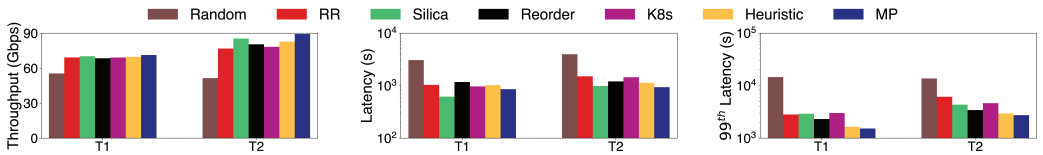


Fig. 13. SLO violation rate of different disk group schedulers (Section 7.4).



(a) Synthesized traces.



(b) Real traces. T1 for trace 1 (bursty restore requests) and T2 for trace 2 (bursty backend requests).

Fig. 14. Throughput, latency, and 99^{th} latency results of different disk group schedulers (Section 7.4).

scheduler. Note that for light workloads (i.e., 0.0625 to 2 req/s), the throughput is limited by the intensity of requests rather than the scheduling, and the latency is worse since MP chooses to spin fewer disk groups to save power (see Figure 12) while maintaing similar throughput. Similar results are shown in Figure 14(b), where MP exhibits 1.08× throughput, 1.52× latency and 1.45× 99^{th} latency improvements under real traces.

Next, we evaluate the SLO performance under heavy workloads (4 req/s and more) since violations are rare under light ones. As shown in Figure 13, MP provides considerations for both SLO and fairness which guarantees all urgent request SLOs even under 16 req/s workload. Silica does not consider request priority characteristics which results in 95.1% violation in urgent requests. The above experiments show that MP achieves better performance on SLO fairness and can serve more urgent users.

7.5 MasterPlan Deep Dive

We use a series of comparison experiments to validate the effectiveness of our designs in MP.

Inference time. Figure 15 displays the inference time of different schedulers. The heuristic scheduler takes multiple system states as input to compute the scores, thus exhibiting a longer time than RR and K8s schedulers. The significant variance of the heuristic scheduler can be attributed to the fluctuating lengths of the request queues. Despite MP has the longest inference time (12ms on

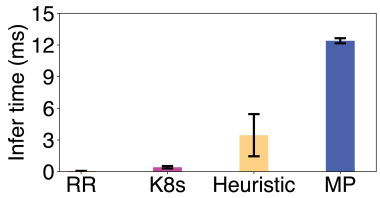


Fig. 15. Inference time of different disk group schedulers (Section 7.5).

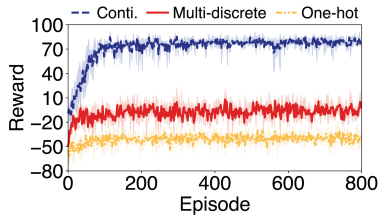


Fig. 16. Learning curves of various action encoding methods (Section 7.5).

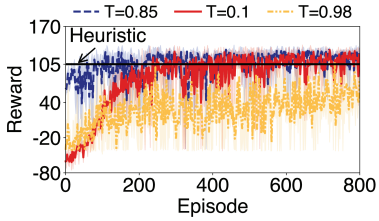


Fig. 17. Learning curves of different action threshold T (Section 7.5).

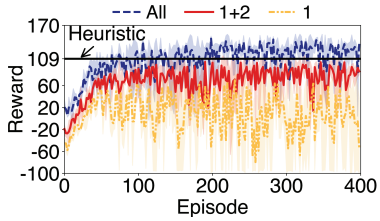


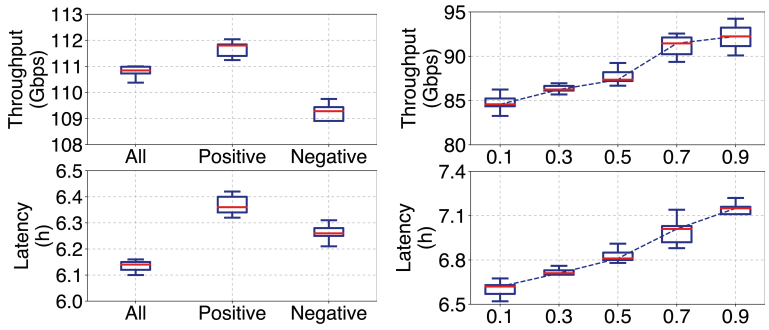
Fig. 18. Learning curves of different state encoding methods (Section 7.5).

average), it is still much shorter than the (at least) second-level scheduling interval. Note that our RL scheduler can scale up linearly under the growth of disk groups due to the continuous action encoding.

Action encoding. Figure 16 shows the reward obtained by the RL agent in each training episode. An agent using the one-hot encoding struggles to achieve positive rewards due to the large action space. While the multi-discrete encoding reduces the action space size, the agent still faces challenges in action exploration. The continuous action encoding allows the agent to explore decision sequences that lead to high rewards up to 85 and facilitates rapid convergence.

Action threshold. Figure 17 shows the training curves of different thresholds used in continuous action encoding, and the horizontal line represents the episode reward of the heuristic scheduler. A lower threshold T increases the likelihood of a power switch among disk groups (Equation (1)). When $T = 0.98$, the agent consistently preserves the current power state, resulting in poor rewards since requests on idle disk groups cannot be served in time. On the contrary, $T = 0.1$ may lead to frequent power switches that results in low rewards at the beginning. Considering the training convergence speed, we opt for a moderate threshold $T = 0.85$, which enables the exploration of reasonable action sequences.

State encoding. Figure 18 depicts the learning curves with different state encoding methods when replaying the same trace, where the number represents the state id listed in Section 5.2. When the agent only observes the disk group power state (denoted as 1), it ignores user request contexts and fails to explore actions to maximize system throughput, resulting in an obvious oscillation characteristic in the training curve. Although not considering latency (denoted as 1 + 2) achieves better throughput performance, it cannot guarantee the SLO due to the lack of request remaining time information, which leads to reward reduction. Only by using the whole state encoding (denoted as All) can MP explore action sequences with higher rewards than the heuristic scheduler.



(a) Different reward functions (b) Different reward weight ratios

Fig. 19. Performance of different rewards (Section 7.5). *Positive/Negative refers to only using the corresponding reward function in Section 5.4. The weight ratio is the ratio between positive and negative rewards.*

Reward function. Figure 19(a) shows the system performance with different reward functions. Using the positive reward function alone enables MP to reach high throughput, however, it does not consider SLO and results in a plethora of request timeouts. The negative reward considers SLO and power draw issues but sacrifices throughput, leading to poor overall system performance. The best performance is achieved through the combination of positive and negative rewards. Additionally, by adjusting the weight ratio between positive and negative rewards, the RL agent can cater to the requirements of different systems. As shown in Figure 19(b), with the rise of weight ratio, throughput improves from 85 Gbps to 92 Gbps while latency also increases from 6.6 to 7.2 hours.

Meta adaptation. We evaluate the effectiveness of adopting the meta adaptation module. We use 20 representative traces as the whole workload dataset, and randomly select one trace as the test workload while using the rest for training MP w/ and w/o meta adaptation. Note that the Ideal scheduler is trained on all 20 traces. We repeated the test for 20 times. As shown in Figure 20, the average performance of MP w/ meta adaptation is 9.8%, 11.0%, 7.7%, and 15.4% higher than results of w/o meta adaptation on reward, throughput, latency and 99^{th} latency. Both MP w/ and w/o meta adaptation outperform the heuristic scheduler due to the efficient RL representation design. Besides, the performance difference between MP w/ meta adaptation and Ideal is small (only 4.3% on throughput). This indicates that the meta adaptation module can effectively improve the performance of MP on unseen traces.

Model retraining. We evaluate the retraining strategy of MP by injecting unseen workloads (with disparate w/r rate and intensity). Figure 21 shows a retraining case where we inject an unseen workload to the system at 10 minutes, and the normalized reward of MP drops significantly. After that, MP triggers the retraining process, and takes only 23.8 minutes to converge. From then on, MP updates the agent model and the observed rewards recover to normal. Generally, retraining process takes less than one hour for convergence, which is acceptable since the time interval between two scheduling actions is nearly tens of minutes (recall in Section 5.3.2). In addition, with the generalization ability of the meta adaptation module, the retraining process happens infrequently (at day-level granularity).

Scheduling interval. We test MP using different scheduling intervals in Figure 22. The system performance gradually decreases with larger intervals since the scheduling granularity becomes coarser and the scheduler cannot capture system status in real-time. Specifically, when increasing

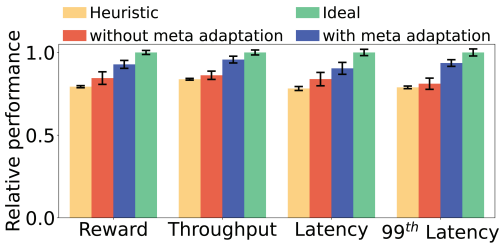


Fig. 20. Relative performance of MP with or without meta adaptation (Section 7.5). *Performance is benchmarked against Ideal's results, the higher, the better.*

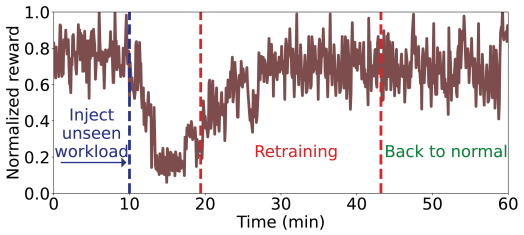


Fig. 21. A case for MP retraining strategy (Section 7.5). *The retraining process takes 23.8 minutes to converge.*

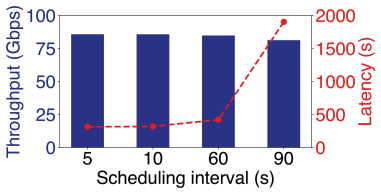


Fig. 22. Throughput and latency with different scheduling intervals (Section 7.5). *Bars are throughput and the dashed line refers to latency.*

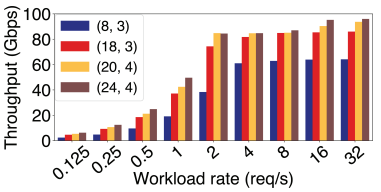


Fig. 23. Throughput with different EC groups (Section 7.6). *Different EC results in varying archive sizes since the disk zone size is fixed.*

the interval from 5s to 90s, we observe a 5.5% reduction in throughput and 6× increase in latency. Consequently, we set 5s as the scheduling interval for MP.

Parameter sensitivity study. In order to evaluate the sensitivity of MP on different parameters, we tune two main parameters for training: the layers of actor-critic network L and the curriculum learning ratio CR (i.e., trace length used for pre-training), and plot the performance heatmap. Note that the results are normalized based on the cell with the best performance. As shown in Figure 25, the throughput and latency results of MP are relatively stable when CR is above 0.4, which is long enough for guiding MP to learn at start. Additionally, the performance becomes worse when L exceeds 3, which is too hard for the agent to convergence. Therefore, we use L=3 and CR=0.4 as the default setting.

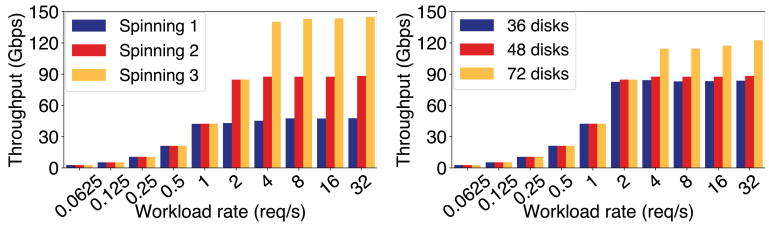
7.6 Varying Hardware Configurations

In this section, we discuss the adaptability of MP under different hardware configurations.

EC group. Figure 23 shows the performance of different EC groups. Note that the smaller the parity number r in (k, r) EC group, the higher the priority of the repair request since it is more likely to cause data loss. We simulate the occurrence of multiple disk failures, and MP prioritizes the urgency of repair requests under a small EC parity number so that data of the failed disk can be recovered as soon as possible.

Spinning disks. Next, we evaluate MP by varying the number of spinning disk groups and the number of disks in a group, which represent different power and cooling design. Figure 24(a) and 24(b) show the throughput of two systems while using the same (20, 4) EC group. We observe that more spinning disks lead to higher system throughput under intensive workload while exhibiting comparable performance under low workload and consuming more power.

The above experiments show that MP can adapt to various archive system configurations.



(a) Different spinning groups (b) Different disks in a group

Fig. 24. System throughput with different number of spinning disks (Section 7.6). *The number of spinning disks is configured by (a) varying the right-provisioning, and (b) the number of disks in a group.*

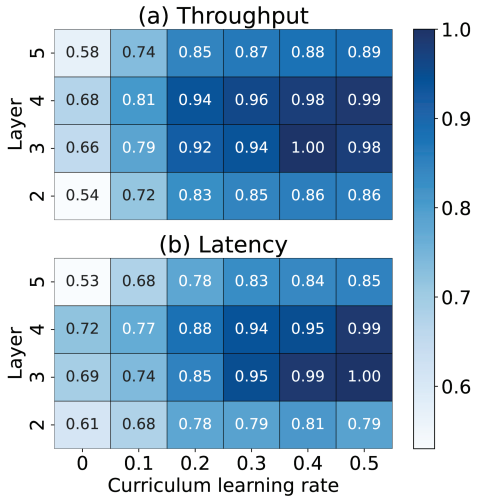


Fig. 25. Parameter sensitivity study on network layer and curriculum learning rate (Section 7.5). *The darker the color, the better the normalized performance.*

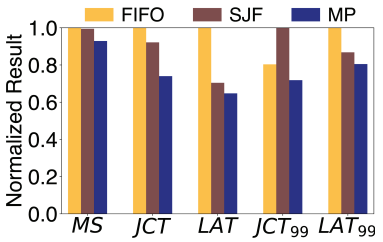


Fig. 26. Performance of different schedulers in a tape-based archive system (Section 7.8). *The smaller the value, the better the performance.*

7.7 End-to-End Evaluation

To validate the end-to-end performance of MP, we conduct a comparative experiment with other two representative schedulers: the heuristic scheduler (scoring-based) and the Silica scheduler (queue-based). These schedulers are evaluated over a 4-hour production trace with bursty urgent restore requests. We show the throughput, number of SLO violations and power results in Figure 27. Note that we present the request size in a 5s window by request type at the bottom. In the beginning, MP discerns the workload being light and thereby reduces the number of spinning disk groups to conserve power consumption. At 1.4 hours, a large number of restore requests arrives and MP agilely adapts to this workload change by intelligently managing spin-up/down actions, yielding a 1.18× throughput improvement compared to the heuristic scheduler. Due to frequent disk group switching, the heuristic results in 2.93× SLO violations. Even worse, Silica exhibits 3.23× SLO violations as it does not consider request urgency during scheduling.

7.8 `MasterPlan` Extension

We also test MP in a tape-based archive system using simulation with production traces and enterprise-level configuration parameters. Figure 26 presents the evaluation results of FIFO,

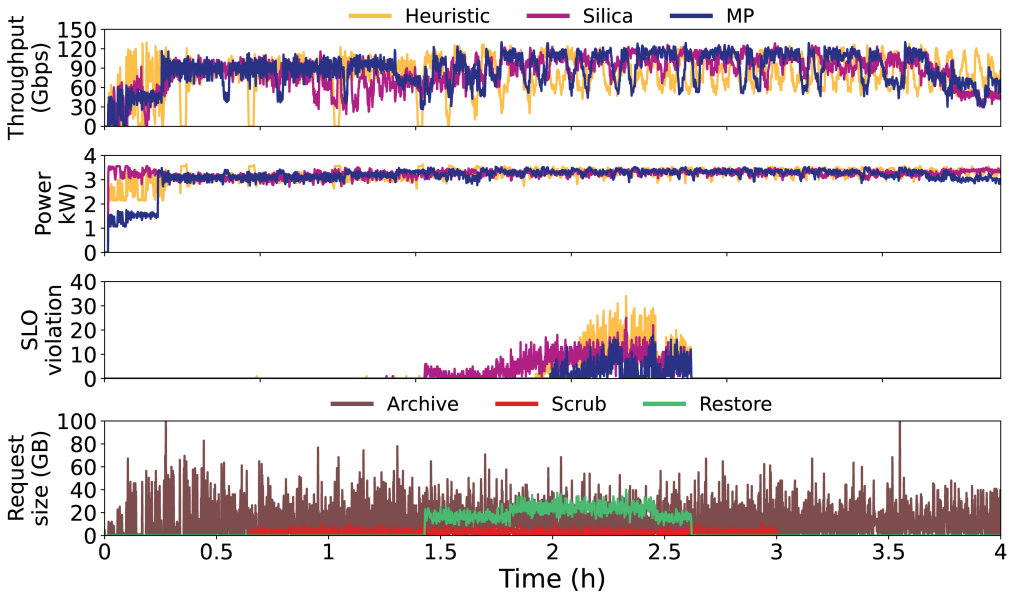


Fig. 27. End-to-end evaluation results of Heuristic (score-based), Silica (queue-based) and MP on a 4-hour trace (Section 7.7). *The request size of each type within a 5s window is shown at the bottom.*

shortest job first (**SJF**), and MP. There are five primary evaluation metrics: MS for makespan, JCT for average job completion time, LAT for average latency, JCT_{99} and LAT_{99} for the corresponding 99^{th} percentiles. The values of each metric are normalized to better compare across different schedulers. The higher the values, the lower performance we get. The FIFO scheduler delivers the worst performance in most metrics except for JCT_{99}. SJF performs worse than FIFO in JCT_{99}, since small requests preempt larger ones in the SJF scheme. In contrast, MP demonstrates the best performance in all metrics, indicating that learning-based schedulers are better suited to adapt to workload shifts and system variations (e.g., waiting tasks and robot positions). Therefore, not only for disks, MP is also a promising solution for scheduling other storage media like tapes in archive systems.

8 Discussion

MasterPlan is currently a research prototype and has not been deployed in production system yet. Currently, we are still evaluating MasterPlan in a small-scale pod, especially its robustness against real-world incidents, for example collateral disk failures. However, we believe that the existing results from the simulator evaluations can prove the effectiveness of MasterPlan scheduling.

The procedure of integrating MP to a real archive storage system can be summarized as follows: (1) Upload a well-trained MP model to each archive server and replace the default heuristic scheduler. Basic scheduler interface (e.g., disk group controller) needs to be compatible with MP; (2) MP periodically polls the archive servers for current system state through timed heartbeat; (3) Once an action has been inferred by MP, sends results to the corresponding archive server for disk group spinning-up/down. Besides, MP stores the trajectory data in the local time-series database (e.g., InfluxDB [28]); (4) If performance degradation is detected, MP retrains its model with the latest trajectory data. Note that during retraining, the archive server can switch to use the heuristic scheduler to ensure system stability.

9 Related Work

9.1 Archive Storage System Design

Archive storage system research has received increasing attention from both industry [2, 5, 11, 16] and academia [14, 15, 32, 47]. Pelican [47] used right-provisioning to save energy costs for cold data workloads. Microsoft has invested years of effort in exploring glass as a storage medium [9, 17]. Facebook used 10,000 blue-ray discs to store cold data [11]. Previous works mainly focus on studying the storage medium and architecture for archive systems, while `MasterPlan` aims to optimize the archive system scheduler.

9.2 RL-based Scheduling Algorithm

RL has been applied in solving various scheduling problems [34, 35, 37, 43, 57]. Decima [37] used an RL agent and graph neural networks to learn a job scheduler. Peng et al. [41] employed RL to optimize the average job completion time in deep learning clusters. AUTO [34] applied an environment-adaptive preference selection method for different optimization goals. Besides, MOCC [35] adopted a multi-objective RL training method for optimizing for different network applications. SchedInspector [57] introduced an RL-based batch job inspector to determine whether a selected job should be executed immediately or delayed. `MasterPlan` differs from the above works in that it focuses on the archive system and optimizes scheduling for its characteristics such as adopting continuous action encoding.

10 Conclusion

This article introduces `MasterPlan`, an RL-based scheduler designed for archive systems. The design of `MasterPlan` mainly focuses on three aspects, the state space and reward function, the action encoding, and the meta adaptation module. Extensive evaluations show that `MasterPlan` effectively increases system overall throughput, reduces SLO violations and saves power under complex and varying workloads. We have open sourced `MasterPlan` along with a full-fledged archive system simulator framework.

References

[1] 2024. Alibaba Cloud OSS Archive. Retrieved from https://www.alibabacloud.com/solutions/backup_archive

[2] 2024. Amazon S3 Glacier Storage Classes. Retrieved from https://aws.amazon.com/cn/s3/storage-classes/glacier/

[3] 2024. Apache HBase. Retrieved from https://hbase.apache.org/

[4] 2024. Azure Archive Storage. Retrieved from https://azure.microsoft.com/zh-cn/solutions/backup-archive/

[5] 2024. Google Cloud Archive Storage. Retrieved from https://cloud.google.com/storage/docs/storage-classes

[6] 2024. IBM Tape Libraries and Tape Automation. Retrieved from https://www.ibm.com/it-infrastructure/tape-library

[7] 2024. Kubernetes Scheduler. Retrieved from https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

[8] 2024. Seagate Cold Data Storage. Retrieved from https://www.seagate.com/sg/en/blog/what-is-cold-data-storage/

[9] Patrick Anderson, Erika Aranas, Youssef Assaf, Raphael Behrendt, Richard Black, Marco Caballero, Pashmina Cameron, Burcu Canakci, Thales de Carvalho, Chatzieleftheriou, et al. 2023. Project silica: Towards sustainable cloud archival storage in glass. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles*.

[10] Jose A. Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. 2019. Rudder: Return decomposition for delayed rewards. In *Proceedings of the 33rd Conference on Neural Information Processing Systems*.

[11] Krish Bandaru and Kestutis Patiejunas. 2015. Under the Hood: Facebook's Cold Storage System. Retrieved from https://engineering.fb.com/2015/05/04/core-data/under-the-hood-facebook-s-cold-storage-system/

[12] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*.

[13] Zahy Bnaya and Ariel Felner. 2014. Conflict-oriented windowed hierarchical cooperative A*. In *Proceedings of the 2014 International Conference on Robotics and Automation*.

[14] James Bornholt, Randolph Lopez, Douglas M. Carmean, Luis Ceze, Georg Seelig, and Karin Strauss. 2016. A DNA-Based archival storage system. In *Proceedings of the 21st USENIX Symposium on Operating Systems Design and Implementation*.

[15] Renata Borovica-Gajić, Raja Appuswamy, and Anastasia Ailamaki. 2016. Cheap data analytics using cold storage devices. In *Proceedings of the 42th Very Large Data Base Endowment*.

[16] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. 2011. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23th ACM Symposium on Operating Systems Principles*.

[17] Andromachi Chatzieleftheriou, Ioan Stefanovici, Dushyanth Narayanan, Benn Thomsen, and Antony Rowstron. 2020. Could cloud storage be disrupted in the next decade?. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems*.

[18] Pim De Haan, Dinesh Jayaraman, and Sergey Levine. 2019. Causal confusion in imitation learning. In *Proceedings of the 33rd Conference on Neural Information Processing Systems*.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*.

[20] Zhuoxuan Du, Jiaqi Zheng, Hebin Yu, Lingtao Kong, and Guihai Chen. 2021. A unified congestion control framework for diverse application preferences and network conditions. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*.

[21] Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*.

[22] Chelsea Finn, Aravind Rajeswaran, Sham Kakade, and Sergey Levine. 2019. Online meta-learning. In *Proceedings of the 36th International Conference on Machine Learning*.

[23] Marek Grzeundefined. 2017. Reward shaping in episodic reinforcement learning. In *Proceedings of the 16th Conference on Autonomous Agents and Multi-Agent Systems*.

[24] Ajay Gulati, Arif Merchant, and Peter J. Varman. 2010. mClock: Handling throughput variability for hypervisor IO scheduling. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*.

[25] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E. Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda.. 2020. Protean: VM allocation service at scale. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation*.

[26] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep reinforcement learning that matters. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*.

[27] Jonathan Ho and Stefano Ermon. 2016. Generative adversarial imitation learning. In *Proceedings of the 30th Conference on Neural Information Processing Systems*.

[28] InfluxData. 2024. InfluxDB. Retrieved from https://www.influxdata.com/

[29] Syed M. Iqbal, Haley Li, Shane Bergsma, Ivan Beschastnikh, and Alan J. Hu. 2022. CoSpot: A cooperative VM allocation framework for increased revenue from spot instances. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC'22)*.

[30] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A deep reinforcement learning perspective on internet congestion control. In *Proceedings of the 36th International Conference on Machine Learning*.

[31] Vijay Konda and John Tsitsiklis. 1999. Actor-critic algorithms. In *Proceedings of the 13rd Conference on Neural Information Processing Systems*.

[32] Sergey Legtchenko, Xiaozhou Li, Antony Rowstron, Austin Donnelly, and Richard Black. 2016. Flamingo: Enabling evolvable HDD-based near-line storage. In *Proceedings ofd the 14th USENIX Conference on File and Storage Technologies*.

[33] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. 2018. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*.

[34] Xu Li, Feilong Tang, Jiacheng Liu, Laurence T. Yang, Luoyi Fu, and Long Chen. 2021. AUTO: Adaptive congestion control based on multi-objective reinforcement learning for the satellite-ground integrated network. In *Proceedings of the 2021 USENIX Annual Technical Conference*.

[35] Yiqing Ma, Han Tian, Xudong Liao, Junxue Zhang, Weiyan Wang, Kai Chen, and Xin Jin. 2022. Multi-objective congestion control. In *Proceedings of the 17th European Conference on Computer Systems*.

[36] Peter Macko, Xiongzi Ge, John Haskins Jr., James Kelley, David Slik, Keith A. Smith, and Smith Maxim G. 2017. SMORE: A cold data object store for SMR drives. In *Proceedings of the 34th International Conference on Mass Storage Systems and Technologies*.

[37] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*.

[38] Markus Mäsker, Lars Nagel, Tim Süß, André Brinkmann, and Lennart Sorth. 2016. Simulation and performance analysis of the ECMWF tape library system. In *Proceedings of the 16th International Conference for High Performance Computing, Networking, Storage and Analysis*.

[39] Eduardo F. Morales and Claude Sammut. 2004. Learning to fly by combining reinforcement learning with behavioural cloning. In *Proceedings of the 21st International Conference on Machine Learning*.

[40] Yudha P. Pane, Subramanya P. Nageshrao, and Robert Babuška. 2016. Actor-critic reinforcement learning for tracking control in robotics. In *Proceedings of the IEEE 55th Conference on Decision and Control*.

[41] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, Chen Meng, and Wei Lin. 2021. DL^2: A deep learning-driven scheduler for deep learning clusters. *IEEE Transactions on Parallel and Distributed Systems* 32, 8 (2021), 1947–1960.

[42] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*.

[43] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T. Kalbarczyk, Tamer Basar, and Ravishankar K. Iyer. 2023. AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems. In *Proceedings of the 2023 USENIX Annual Technical Conference*.

[44] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding with unsupervised learning. *Technical Report, OpenAI* (2018). https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[45] Simon Schmitt, Matteo Hessel, and Karen Simonyan. 2020. Off-policy actor-critic with shared experience replay. In *Proceedings of the 37th International Conference on Machine Learning*.

[46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. arXiv:1707.06347. Retrieved from https://arxiv.org/abs/1707.06347

[47] Balakrishnan Shobana, Black Richard, Donnelly Austin, Paul England, Glass Adam, Harper Dave, Legtchenko Sergey, Ogus Aaron, Peterson Eric, and Rowstron Antony. 2014. Pelican: A building block for exascale cold data storage. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*.

[48] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies*.

[49] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.

[50] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144.

[51] Ya Su, Youjian Zhao, Chenhao Niu, Rong Liu, Wei Sun, and Dan Pei. 2019. Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

[52] Christine Taylor. 2021. What Is Cold Data Storage? Storing Cold Data in the Cloud. Retrieved from https://www.enterprisestorageforum.com/management/cold-cloud-data-storage/

[53] Chen Tessler, Yuval Shpigelman, Gal Dalal, Amit Mandelbaum, Doron Haritan Kazakov, Benjamin Fuhrer, Gal Chechik, and Shie Mannor. 2022. Reinforcement learning for datacenter congestion control. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st Conference on Neural Information Processing Systems*.

[55] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems*.

[56] Zhengxu Xia, Yajie Zhou, Francis Y. Yan, and Junchen Jiang. 2022. Genet: Automatic curriculum generation for learning adaptation in networking. In *Proceedings of the ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*.

[57] Di Zhang, Dong Dai, and Bing Xie. 2022. SchedInspector: A batch job scheduling inspector using reinforcement learning. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*.

Received 21 July 2024; revised 31 October 2024; accepted 4 December 2024