A Bring-Your-Own-Model Approach for ML-Driven Storage Placement in Warehouse-Scale Computers

Chenxi Yang [1] **Yan Li** [2] **Martin Maas** [3] **Mustafa Uysal** [2] **Ubaid Ullah Hafeez** [2] **Arif Merchant** [2] **Richard McDougall** [2]

Abstract

Storage systems account for a major portion of the total cost of ownership (TCO) of warehouse-scale computers, and thus have a major impact on the overall system's efficiency. Machine learning (ML)-based methods for solving key problems in storage system efficiency, such as data placement, have shown significant promise. However, there are few known practical deployments of such methods. Studying this problem in the context of real-world hyperscale data centers at Google, we identify a number of challenges that we believe cause this lack of practical adoption. Specifically, prior work assumes a monolithic model that resides entirely within the storage layer, an unrealistic assumption in real-world deployments with frequently changing workloads. To address this problem, we introduce a cross-layer approach where workloads instead "bring their own model". This strategy moves ML out of the storage system and instead allows each workload to train its own lightweight model at the application layer, capturing the workload's specific characteristics. These small, interpretable models generate predictions that guide a co-designed scheduling heuristic at the storage layer, enabling adaptation to diverse online environments. We build a proof-of-concept of this approach in a production distributed computation framework at Google. Evaluations in a test deployment and large-scale simulation studies using production traces show improvements of as much as 3.47× in TCO savings compared to state of the art baselines.

1 Introduction

Storage systems comprise a large part of data centers' total cost of ownership (TCO). Even small improvements in storage system efficiency can have a major impact on the overall costs. Improvement as low as 1% represents a large amount in the context of hyperscale data centers, which see billions of dollars of investment (Molla, 2018). Data placement – e.g., deciding whether a file should be stored on hard disk (HDD) or solid state drives (SSD) – is an important decision impacting the efficiency and costs in data center storage systems. This problem is also known as storage tiering and has been the subject of a large amount of research (Kim et al., 2014; Dulloor et al., 2016; Saxena et al., 2012). Currently, there are two approaches to this problem:

- **Heuristics**, such as greedily allocating data to SSDs until capacity is reached and using HDDs for overflow (Albrecht et al., 2013; Yang et al., 2023b; Eytan et al., 2020; Yang et al., 2013; 2022). These heuristics are deployed and represent today's state-of-the-art. They are fast and interpretable, but perform suboptimally when SSD capacity is limited.

- **Machine Learning (ML)** approaches that leverage real-world workload information (Zhou & Maas, 2021). Few of these approaches are practically deployed due to considerations such as run-time overhead, adaptability, or decomposition challenges, and risks associated with a model becoming a single point of failure (Maas, 2020; Paleyes et al., 2022).

To understand the challenges behind adoption of ML in real-world scenarios, we analyzed our real-world production systems at Google. Data centers run a wide range of workloads with vastly different characteristics (Figure 1). Workloads arrive and evolve at a high rate, and data access patterns are highly dynamic and application-specific. Data centers deal with this issue through *abstraction layers*, such as the application, storage or hardware layer: For example, the application layer does not need to worry about the specifics of the hardware, and the storage layer does not need to know any details about the inner workings of each application. This enforces separation of concerns and allows these components to operate and evolve independently.

We posit that this is the key challenge behind existing proposed ML methods. Existing ML works mostly treat end-to-end data placement as one problem and assume a monolithic model deployed within the storage system (Liu et al., 2020; Singh et al., 2022; Zhou & Maas, 2021; Kaler & Toshniwal, 2023). Such a model might be trained on file names or

[1]UT Austin. Work was done while at Google. [2]Google [3]Google DeepMind. Correspondence to: Yan Li <elliotli@google.com>.

Proceedings of the 8th MLSys Conference, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).

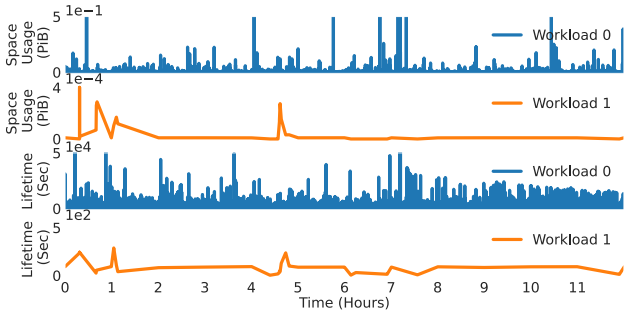


Figure 1: Workloads show vastly different storage patterns.

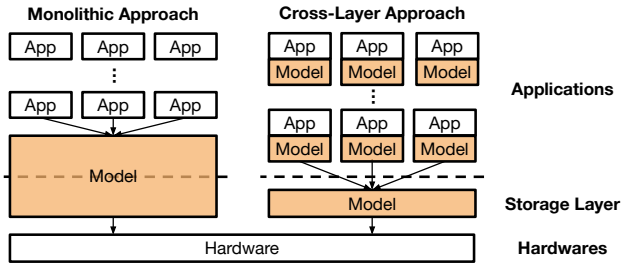


Figure 2: Conceptual overview of the monolithic approach vs. the cross-layer approach.

common application behavior (Zhou & Maas, 2021). While this approach works in simulation, it breaks the separation of concerns, which is problematic in real-world large-scale systems. For instance, changes to a major application that affect file names would need to trigger a retraining of the model at the storage layer, which needs to roll out changes at a much lower velocity than the workloads.

We introduce a "Bring-Your-Own-Model" (BYOM) design that embraces the multi-layer characteristic of storage systems and presents a practical solution to these problems by combining cheap and interpretable ML models at the application layer with a custom algorithm that leverages their predictions at the storage layer. Instead of a single large ML model, we build smaller models for individual workloads, which produce hints that the storage layer can utilize to place the workloads' data (Figure 2).

Evaluating this approach at scale presents a chicken-and-egg problem. Porting a sufficiently large number of complex workloads to this approach is a large investment and requires the approach to be in place, but putting the approach in place requires experimental validation with a large range of workloads. We side-step this issue by focusing on a particular class of workloads that are written in a distributed data processing framework similar to Apache Beam (Nurmamat kyzy et al., 2012) or Apache Spark (Zaharia et al., 2012). This framework is used for a wide range of workloads in our fleet, including log processing, simulations, streaming applications, and a variety of ML workloads (which may use accelerators). By targeting tiering of intermediate files generated by these workloads, we can use large-scale historical traces and evaluate our approach across a highly diverse set of applications (Figure 1), without having to port each of them manually. Note that cumulatively, these files can account for a significant portion of all storage resources in a data center (up to 35% in some clusters).

We first present a headroom analysis to understand the potential upside from ML over traditional heuristics. We formulate the data placement problem into an Integer Linear Programming (ILP) problem and use a solver to determine optimal placement decisions. We find that these optimal decisions can achieve 5.06× the cost savings of a state-of-the-art heuristic approach (but require clairvoyant knowledge).

Prior work has proposed imitation learning against such an oracle (Liu et al., 2020). However, we find that this approach does not work in our deployment, since the model does not only need to make decisions for individual workloads but *adapt* to an environment that is changing due to external factors (e.g., varying load patterns and workloads arriving or leaving). Our BYOM design tackles this *adaptability problem*. At the application layer, we analyze the data properties that contribute to the optimal placement. We then design a model to predict a ranking based on these properties, which is independent of online fluctuations of the environment and other applications. Finally, we design an adaptive algorithm at the storage layer to select the data to place on SSD based on the model predictions from all applications and system feedback.

We instantiate this design in the context of a production data processing framework at Google. We show its practicality by developing a prototype of our approach and running it in a test deployment. We also perform an extensive simulation study based on real-world production traces from Google's data centers. We show that our approach can lead to an additional 3.22% TCO savings, more than 3.47× the savings from the production baseline. We summarize our contributions as follows:

- Within a real-world production setup, we investigate ML for storage placement from a new perspective, with a focus on practicality in production settings.
- We design and instantiate a novel cross-layer approach, combining ML and heuristics that can adapt across workloads and external factors in data centers.
- We prototype the proposed ML integration to show its realizability in a real production codebase.
- We evaluate our method at scale with production traces and achieve 3.47× TCO savings over SOTA baselines.

We first present background on storage systems and production constraints (Section 2), and formulate our optimiza-

tion problem with baselines (Section 3). We then discuss our approach (Section 4), introducing our ML method and scheduling algorithm. We next show detailed prototype and large-scale simulation studies (Section 5). Finally, we discuss related work (Section 6) and conclude (Section 7).

2 BACKGROUND

2.1 Storage for Data Processing Frameworks

Modern data processing frameworks, such as Apache Beam, structure their computations as data flow graphs (Figure 3, left). Each node (or *step*) within the graph represents a computation step. Edges represent the flow of data. Computations are highly parallel, and a distributed framework spawns *workers* to execute tasks. A worker is a process that runs on a server. Data is generally passed between workers through *shuffle jobs*. A shuffle job is generated when the execution of the workflow reaches a step or operation that necessitates the exchange of information. As an example, *GroupByKey* is a common operation across frameworks that generates one or more shuffle jobs. During a shuffle job, workers write their data as *intermediate files* to a distributed file system (Shvachko et al., 2010; Calder et al., 2011; Ghemawat et al., 2003) and read it in subsequent steps. The access patterns to the files depend on the specifics of the computation, such as filtering, grouping, or sorting. One shuffle job can operate on multiple intermediate files.

2.2 SSD/HDD Tiering and its Trade-Offs

Storage cost falls into several different categories: 1) the amount of storage (e.g., in GiB) occupied by the data, 2) wearout of devices such as SSD, which degrade with every write, and 3) the amount of I/O operations (i.e., read or write requests to a storage device per unit time sustained by the device). SSDs and HDDs have different trade-offs among all three dimensions. SSDs provide much larger amounts of I/O with a higher cost per GiB and write-induced wearout. In contrast, HDDs are ideal for large amounts of data and long sequential access patterns that introduce few I/O operations. At the same time, SSDs are ideal for random, small accesses – if the resulting wearout can be tolerated. In practice, intermediate files in data processing pipelines can fall into either category (or, more commonly, in between), which makes the data placement problem challenging.

2.3 Production Requirements and Limitations

Data centers accommodate a vast array of workloads with diverse behavior patterns. Employing a single model in the storage layer to jointly learn all workloads introduces a single point of failure, and requires the model to be large and complex, and thus expensive and difficult to interpret. Further, this approach requires all input features to be reliably delivered to the storage system, when some of the most predictive features are workload-specific (Zhou & Maas, 2021). Finally, In hyperscale data centers, workloads exhibit significantly faster rates of change than the update cycles of storage systems. This causes a dilemma: 1) Rolling out the model with the storage system means it is stale by the time it reaches production; 2) updating the model independently of the storage system means that it is not tested as rigorously as the rest of the system, thus becoming its weakest link.

To address these problems, we propose a more granular approach where each workload has its own dedicated model to produce a *hint*, which is then reliably passed to the storage system. This hint indicates, for example, how well a file can be cached. Since workloads "*bring their own model*", models evolve at the velocity of the workload rather than the storage system. Because the models are smaller, they are cheaper and more interpretable. They are distributed across many workloads hence they can use more features and are more robust: a model failure only affects one workload. In this work, we focus on optimizing data placement for workloads built on a particular data processing framework, but our cross-layer design is general.

Training a model offline and deploying it online is challenging, since a static model cannot adapt to evolving workload patterns, which are prevalent in real-world scenarios. To address this issue, we present an adaptive strategy that utilizes online observations to inform placement decisions.

Models can introduce non-trivial overheads. For example, prior work suggested using Transformers, which are known for their superior learning capabilities but can be computationally expensive, incurring prediction latency costs of approximately 99ms (Zhou & Maas, 2021). To balance performance and efficiency, we leverage gradient boosted trees as the model, providing a compromise between lightweight, low-performance models and powerful, expensive models.

2.4 Google's Production Setup

Our fleet consists of clusters of up to O(10,000) machines. Dedicated storage servers handle all requests to the distributed storage system. SSD and HDD devices are hosted by different servers due to different device form factors. While regular servers can also have locally attached storage, it is mainly used by the local OS and not for compute data.

A global distributed file system is used to store files. The system provides a client library that resides on all compute servers and communicates with storage servers. Data processing frameworks running on top of compute servers use this library to store and retrieve files. SSD tiering is handled by a service running on a dedicated set of servers that analyze the client traffic and make caching decisions (admit to/delete from SSD, evict to HDD,...). They proxy requests

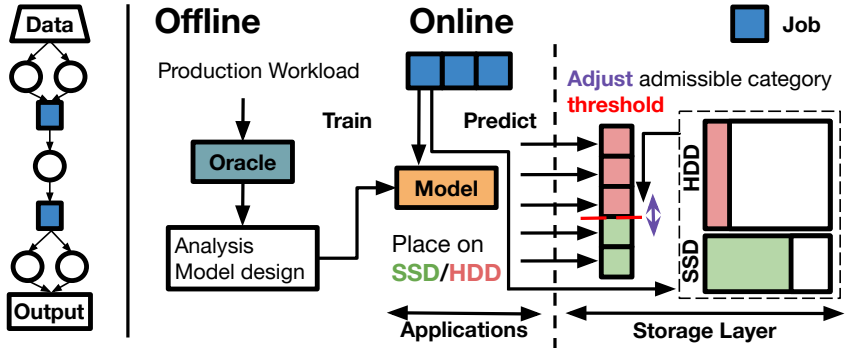


Figure 3: **Left:** Data flow graph in a data processing framework. Data is processed in parallel and its jobs create intermediate files (blue) which are inputs for the next processing step. **Right:** Approach Overview. We analyze production workloads offline for model design and training. Online, each application's model predicts job properties and passes the prediction to the storage layer for job placement.

to the appropriate storage servers. This work focuses on the SSD/HDD tiering decisions within these caching servers. More details are shown in Appendix A.

3 Problem Formulation & Baselines

We now provide a concrete definition of the data placement problem. Our basic data placement unit is a shuffle job. We track four attributes for each job: (start time, lifetime, job size, cost). Job size is measured in bytes. A cluster has an SSD capacity. The placement algorithm produces a mapping: job $\rightarrow \{\text{SSD}, \text{HDD}\}$.

Resource savings from SSD tiering are relative to a baseline where all files are stored on HDD. Our main optimization criterion is storage resources. While SSD tiering may also reduce CPU/RAM resource consumption of individual jobs and improve their tail latency, those improvements are *opportunistic* and not the target of our work. Since workloads are written assuming HDD storage, they do not rely on these speed-ups. Further, we found that *aggregate* CPU/RAM resource consumption across all jobs is not significantly affected by the tiering technique that is used.

We evaluate resource savings from two perspectives. First, we measure the reduction in HDD I/O from moving jobs to SSDs. This frees up HDD I/O for operations that cannot (or should not) be moved off HDDs, such as accesses to cold data. We quantify this reduction using a metric called *Total Cost of I/O* (TCIO), where a value of 1.0 represents the amount of I/O that a standard HDD can sustain per second. Jobs running on SSDs have a TCIO of zero. Our TCIO calculation accounts for caching effects, such as the DRAM cache present alongside HDDs in each server. In our system, I/Os that are served from cache do not reach the disks, and small write operations are grouped into 1 MiB chunks before reaching the disks. This ensures that TCIO reflects the true workload pressure on the disks.

Second, we measure overall monetary savings. We define *Storage Total Cost of Ownership* (TCO) as the total expenditure associated with acquiring, operating, and maintaining a storage system. TCO is calculated separately for HDDs and SSDs due to the different nature of these devices. Substitute DEV for HDD or SSD below to get the respective definition:

$$
\begin{aligned}
\text{TCO}^{\text{DEV}} &= \text{cost}^{\text{DEV}}_{\text{byte}} + \text{cost}^{\text{DEV}}_{\text{network}} + \text{cost}^{\text{DEV}}_{\text{server}} \\
&\quad + \text{cost}^{\text{DEV}}_{\text{specific}} \\
\text{cost}^{\text{DEV}}_{\text{byte}} &= \text{byte_cost}^{\text{DEV}} \cdot \text{size} \cdot \text{duration} \\
\text{cost}^{\text{DEV}}_{\text{network}} &= \text{network_cost_rate} \cdot \text{IO_throughput}^{\text{DEV}} \\
&\quad \cdot \text{duration} \\
\text{cost}^{\text{HDD}}_{\text{server}} &= \text{server_cost_rate}^{\text{HDD}} \cdot \text{TCIO} \cdot \text{duration} \\
\text{cost}^{\text{SSD}}_{\text{server}} &= \text{server_cost_rate}^{\text{SSD}} \cdot \text{IO_throughput}^{\text{SSD}} \\
\text{cost}^{\text{HDD}}_{\text{specific}} &= \text{device_cost_rate}^{\text{HDD}} \cdot \text{TCIO} \cdot \text{duration} \\
\text{cost}^{\text{SSD}}_{\text{specific}} &= \text{wearout_cost_rate}^{\text{SSD}} \cdot \text{total_written_bytes}
\end{aligned}
$$

where $*\text{_cost_rate}$ denotes conversion rates to dollar cost; $\text{cost}^{\text{DEV}}_{\text{byte}}$ denotes the cost of storing one byte for one second on a device; TCIO, size, and duration denote a job's TCIO need, storage footprint, and duration (for example, if a job has a TCIO of 2, the job would need two HDDs to run). The $\text{cost}^{\text{DEV}}_{\text{network}}$ is a value derived from the data center total network cost of transmitting data at the aggregated throughput for the duration of the job. The network cost is largely constant and independent of the device, but we include it to avoid overestimating the impact of the other costs on the overall TCO. $\text{cost}^{\text{HDD}}_{\text{server}}$ and $\text{cost}^{\text{HDD}}_{\text{specific}}$ cover the cost of the servers and HDDs.

In practice, we found that the server cost for running a job on SSD correlates with the bytes transmitted. Because all SSD devices have a limit on the amount of Program/Erase

(P/E) cycles and each write causes a loss in monetary value, the $\text{cost}^{\text{SSD}}_{\text{specific}}$ is included to cover these wearout costs. $\text{wearout_cost_rate}^{\text{SSD}}$ is calculated from the specific SSD drive model's total bytes written rating.

3.1 Oracle: Optimal Solution Based on Solver.

To better understand the headroom that is available if we achieve perfect data placement, we design an oracle. It is an upper bound on the best solution, but impossible to implement. The oracle policy is based on using an Integer Linear Programming (ILP) solver and clairvoyant knowledge — that is, assuming we know the future access pattern. We formulate the placement problem as an ILP problem:

$$
\begin{aligned}
& \max \sum_{i \in l} x_i (c_i^{\text{HDD}} - c_i^{\text{SSD}}) \\
\text{subject to:} \quad & x_i \in \{0, 1\}, \forall i \in [0, N] \\
& p_i(t) = x_i s_i, \forall i \in [0, N], a_i \leq t \leq e_i \\
& \sum_{i \in [0,N], a_i \leq t \leq e_i} p_i(t) \leq M, \forall t \in T \\
& T = \max\{e_i : i \in [0, N]\}
\end{aligned}
$$

The SSD space limit is M and we assume the HDD space is infinite due to its lower cost per GiB. $X = \langle x_0, \ldots, x_N \rangle$ is a sequence of arriving jobs. Job i, represented by variable x_i, arrives at time a_i, ends at time e_i, and it needs s_i space with c_i^{SSD} cost to put on SSD, c_i^{HDD} cost to put on HDD. Oracle optimization can either optimize for TCIO or TCO. We use a binary variable, x_i, to denote the data placement decision, $x_i = 1$ if i is put on SSD, and $x_i = 0$ if i is put on HDD. Once placed, a job x_i runs from a_i to e_i and $p_i(t) = x_i s_i$ represents the job's SSD consumption at time $t \in [a_i, e_i]$. Now the problem becomes maximizing the savings by placing jobs on SSD under space constraints (SSD space is limited): We run the above ILP with historical production workload data and find the optimal placement decisions that save the maximum amount of cost. The solver is optimal because it has clairvoyant knowledge. The clairvoyant knowledge includes information that is not available at a job's placement decision time in practice: 1) The solver knows the global job ranking in terms of cost savings and would prioritize putting high cost saving jobs onto SSD. 2) The solver knows the workload patterns and places jobs that would monopolize SSD resources on HDD instead.

In addition to clairvoyant knowledge, the oracle needs a fixed SSD capacity limit for optimal placement. However, we do not have such knowledge ahead of time as the data center is shared between many jobs and free SSD capacity fluctuates over time. Thus, a solution that can apply under varying SSD capacities is needed. We now list our baselines.

3.2 FirstFit: Static Placement.

Production systems commonly perform HDD/SSD tiering using FIFO or LRU-style heuristics (Yang et al., 2023a; Eytan et al., 2020). We implement a representative instance of this approach. We try to place jobs on SSD in the order of their start times, checking jobs' peak space usage and only placing jobs on SSD that fit in the available SSD capacity. This optimizes TCIO when unlimited SSD is available but can significantly increase TCO when SSD capacity is limited or expensive.

3.3 Heuristic: Practical Adaptive Placement.

Recently, heuristics that can dynamically adapt to workloads have been introduced, striking a balance between dynamically learning workload behavior and avoiding the practicality issues in Section 2.3. We emulate the state-of-the-art placement approach from (Yang et al., 2022). It focuses on a slightly different problem (SSD read cache admission), but can be adapted for our placement task. We use this approach as a stand in for the closest practical approach to a learning-based baseline.

The approach starts from a set of categories associated with storage requests and then constructs a per-category admission policy based on dynamic behavior. In our experiments, we use the job's *ID* as the category. For each job category, the approach measures space usage and TCO savings. We rank the categories by their TCO savings and add categories into an *admission set* until the selected category's historical space usage reaches the SSD capacity. When a new job arrives, we decide to place it on SSD if it belongs to the admission set. Otherwise, the job is placed on HDD.

3.4 ML Baseline: Lifetime Prediction-Based.

We include another closely related ML approach (Zhou & Maas, 2021), which models storage problems in data centers as distribution prediction problems. We follow the paper's SSD/HDD tiering case study to predict the mean (μ) and standard deviation (σ) of a file lifetime. Files with a predicted lifetime ($\mu + \sigma$) shorter than the specified time-to-live (TTL) are admitted to SSD. To mitigate mispredictions, we evict any file residing in the SSD for longer than $\mu + \sigma$, as in the paper.

4 Hybrid Learning Approach

A common approach to ML-driven systems is to train a model that learns to make decisions, such as whether to place a file on SSD or HDD – e.g., via imitation learning (Liu et al., 2020). However, data centers are highly dynamic environments and the optimal decision depends on external factors such as the available amount of SSD at a given point in time. Thus, a model would need to jointly learn the exter-

nal environment *and* the workload, which is challenging and not deployable, requires a possibly prohibitive amount of training data, and may be brittle when facing new scenarios.

To address this and our other challenges (Section 2.3), we allow each workload to "bring their own model". We propose a *cross-layer* learning approach that uses the model only to predict a *proxy* for workload-specific characteristics and then co-designs a storage-level heuristic that turns the predictions into decisions for the current environment. This design ensures that each workload's I/O patterns is treated as independent, preventing one workload from impacting the I/O costs of others. Specifically, we want to design a proxy that correlates with a job's TCO savings. The proxy allows predicted results to directly represent how a job's placement contributes to end-to-end cost savings.

We call this proxy metric "importance" and train a categorical pointwise ranking model (*category model*) to learn the job's *importance ranking* (Figure 3 right). Each category represents a particular level of importance. A higher ranking indicates a more important job – placing it on SSD saves more cost. When making a placement decision, we query the model for an *importance ranking category* for a new job. We then run an adaptive category selection algorithm with dynamic feedback from the storage layer to decide which importance ranking categories to admit onto SSD.

4.1 Features

We train our model on application-level features from production traces. The features span execution metadata, job timestamps, allocated resources, and historical system metrics, which reflect how jobs are processed in our setting.

Each shuffle job has three main steps: data writing, sorting, and data retrieval. Workers first write raw intermediate files, which are then organized into sorted intermediate files by sorters. Finally, workers retrieve the required data from the sorted files back into memory. These steps can overlap in time, depending on the job setup. To represent job I/O density, we utilize internal job-related information from the framework, which breaks down data into buckets that are assigned to workers for execution. Buckets help distribute the workload evenly and improve parallel writing efficiency.

The features we choose (Table 2 in Appendix B) capture how these steps are executed (allocated resources, execution metadata). Execution metadata is formatted as strings that detail execution-related names, paths and targets. Key elements are separated by non-alphanumeric characters. We treat execution metadata as a sequence of substrings representing the key elements (Table 3 in Appendix B). Since executions may run periodically, we also include the weekday and hour of the day of a job's start time. Allocated resource information represents resources assigned to the job by the cluster scheduler, before it starts execution. However, specific details regarding resource distribution, such as the assignment to SSD or HDD, are not determined at this stage. In addition, we also incorporate properties of previously completed jobs from the same user's pipelines, including the past TCIO, job lifetime, and size. For a detailed description of these features, please refer to Appendix B.

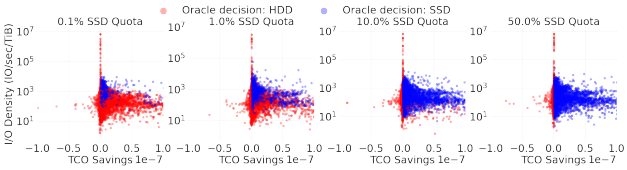


Figure 4: I/O density and TCO savings of each job (color shows oracle placement decision when optimizing for TCO). Tested under different SSD quota.

4.2 Model Design

We use gradient boosted trees instead of neural networks (which are much more expensive and less interpretable) or lookup tables (which sacrifice accuracy). We build on the Yggdrasil Decision Forests framework (YDF) (Guillame-Bert et al., 2023). Our model is trained on the features in Section 4.1.

Quantifying Job Importance. Our goal is for the model to determine each job's *importance*, which is equivalent to the expected cost savings. We first design a way to represent this *importance*. We examine the oracle placement (from Section 3) under different SSD capacities. We expect the most important jobs to be admitted by the oracle even under extremely limited SSD capacity, and as the SSD capacity increases, less important jobs are admitted.

For each job, we compute binary placement decisions (SSD/HDD) from the oracle with different SSD capacity limits. In Figure 4, we show how oracle decisions correlate with TCO savings and *I/O density*, which denotes the total I/O across the job lifetime divided by its maximum storage footprint. As the SSD capacity increases, more jobs with lower I/O density are chosen for SSD. Since the oracle optimizes for TCO, jobs with negative TCO savings if put on SSD should never be selected. Further, if two jobs have the same I/O, small and short-lived jobs are preferred as they occupy less SSD capacity for less time. This suggests that predicting the *sign of TCO savings* and *I/O density* is a good proxy for job importance: negative TCO saving jobs are least important; jobs with higher I/O density are more important.

Label Design. Predicting precise values of these properties has been shown to be challenging, even in theoretical works – e.g., lifetime in (Zhou & Maas, 2021). Rather than treating the importance prediction problem as regression,

we choose a categorical pointwise ranking model, which groups jobs with similar (TCO savings, I/O density) into the same category (i.e., importance ranking class). The idea of framing the output prediction issue as a classification task is commonly adopted in the fields of image and audio analysis (Kang & Park, 2020; Lee et al., 2009).

To pick the specific categories, we need to take into account that our goal is for these categories to provide a ranking of the "importance" of placing each job on SSD. First, negative TCO saving jobs should have the lowest ranking and we set aside one category specifically for these jobs. For the remaining categories, our goal is to cluster jobs by their I/O density. We found that both linear and logarithmically spaced categories would result in a heavily imbalanced data set (Figure 4). Therefore, we choose the categories so that they evenly divide the training set by I/O density (e.g., top 10%, top 20%, top 30%, etc.). For a model with N categories, this results in the following category labels, given TCO savings $x.m$, I/O density $x.n$, and training set size D:

$$\mathrm{C}(x) = \begin{cases} 0, & \text{if } x.m < 0 \\ k, & \text{if } x.n \in (\text{top } \frac{N-k}{N-1} * D, \text{top } \frac{N-k-1}{N-1} * D] \\ & \quad \text{and } x.m \geq 0 \end{cases}$$

4.3 Adaptive Category Selection Algorithm

We now discuss how our cross-layer design combines the learned category model and a heuristic-based algorithm for online data placement. As mentioned in Section 4, the model's category prediction is independent of the SSD capacity. How to select the categories to place on SSD under varying SSD capacities is non-trivial. A simple approach is to fix the admittable categories and always only place jobs predicted in these categories onto SSD. However, as shown in Figure 4, when we have larger SSD capacities, we want to dynamically admit more jobs compared to the smaller capacities case – that is, more categories.

Algorithm Overview. Our algorithm makes its admission decisions based on real-time feedback regarding SSD utilization. When observing limited SSD capacity, we gradually decrease the number of categories to admit. Otherwise, we admit more categories. Since our category model predicts the "importance ranking" of jobs, admitting fewer categories naturally leads to admitting the most important jobs. Admitting more categories means that we broaden the admission set by adding less important but still cost saving jobs. We use a sliding category admission threshold to determine which predicted categories get placed on SSD.

SSD Usage Approximation. In cloud data centers, the actual SSD capacity varies among clusters of machines, which is also hard to directly approximate in practice. The criteria for determining whether an SSD is nearly full (i.e. cannot fit more jobs) or not are influenced by workload

Algorithm 1 Adaptive Category Selection Algorithm

input model M_N, $X = \langle x_0, \ldots, x_n \rangle$, t_w, $T_{\text{SPILLOVERTCIO}}$, t_l.
Initialize $t_d = 0$, ACT $= 1$ and $X_h = \emptyset$.
for x_i in X **do**
  Get the current time stamp as x's arrival time $t_i = x_i.t_a$
  if Last admission decision is expired: $t_i \geq t_d + t_l$ **then**
    Update look back window endpoint $w_s, w_e = t_i - t_w, t_i$
    Remove expired jobs:
    $X_h = X_h - \{x_j | x_j.t_a <= w_s\}$
    Update the spillover percentage from X_h:
    $h_{\text{SPILLOVERTCIO}} = P_{\text{SPILLOVERTCIO}}(X_h, t_i)$
    if $h_{\text{SPILLOVERTCIO}} < T_l$ **then**
      ACT $= \max(N-1, \text{ACT}+1)$ **end**
    if $h_{\text{SPILLOVERTCIO}} > T_u$ **then**
      ACT $= \min(1, \text{ACT}-1)$ **end**
    Update decision making time $t_d = t_i$ **end**
  Infer the predicted category $C_i = M_N(x_i.\text{features})$
  if $C_i \geq$ ACT **then**
    Place x_i onto SSD **else**
    Place x_i onto HDD **end**
  Add the job into the observation history $X_h = X_h \cup x_i$
end for

patterns. Thus, we introduce a metric to unify the measurement of SSD capacity usage across clusters and workloads through job behavior observation. Given a sequence of jobs, $X = \langle x_0, x_1, \ldots, x_n \rangle$, we define the *spillover* TCIO *percentage*, $P_{\text{SPILLOVERTCIO}}(X, t)$, to measure the portion of all job x_i's TCIO that is scheduled to be put onto SSD but ends up on HDD due to the fact that the SSD has already reached its full capacity at timestamp t (notation in Table 1):

$$P_{\text{SPILLOVERTCIO}}(X, t) = \frac{\sum_{x_i \in X} \text{SPILLOVERTCIO}(x_i, t)}{\sum_{x_i \in X} x_i.\text{DEV} \cdot x_i.\text{TCIO}_{\text{HDD}}(t)}$$

where $\text{SPILLOVERTCIO}(x, t)$ is the amount of spill over TCIO of a job x at time t, if spillover starts at t_s:

$$\begin{cases} \frac{t - t_s}{t - t_a} \text{TCIO}_{\text{HDD}}(t), & \text{if } t_s \text{ exists and } t_a \leq t_s \leq t \\ 0.0, & \text{Otherwise.} \end{cases}$$

Intuitively, SPILLOVERTCIO measures the amount of the job's intended TCIO savings that are not realized. A large $P_{\text{SPILLOVERTCIO}}$ means that many jobs fail to be scheduled onto SSD, which indicates that the SSDs are nearly full.

Algorithm Design. We now introduce the adaptive category selection algorithm in Algorithm 1 with notation available in Table 1. In the algorithm, we keep track of an observation history X_h, which contains all the jobs starting within a fixed look back window, and calculate the SPILLOVERTCIO within the history, $h_{\text{SPILLOVERTCIO}}$. Then, we adaptively adjust the *admission category threshold* (ACT) based on the observed $P_{\text{SPILLOVERTCIO}}$ – if $P_{\text{SPILLOVERTCIO}}$ is larger than ACT, we increase the threshold to admit fewer categories. One issue of a dynamic control system of this kind is that ACT may change drastically. We combine two mechanisms to smooth the ACT change:

Symbol/Notation	Meaning
$X = \langle x_0, x_1, \ldots, x_n \rangle$	job sequence
x.features	job features available before execution
x.DEV	job scheduled device (0/1 for HDD/SSD)
$t_a, x.t_a$	job arrival time
$t_s, x.t_s$	job spillover time
$t_e, x.t_e$	job end time
$\text{TCIO}_{\text{HDD}}(t)$	job TCIO if put onto HDD until t
$\text{SPILLOVERTCIO}(x, t)$	job spillover TCIO at t
$P_{\text{SPILLOVERTCIO}}(X, t)$	jobs spillover TCIO percent at t
t_w	look back window time length
t_l	admission decision effective time length
$T_{\text{SPILLOVERTCIO}} = [T_l, T_u]$	spillover tolerance range
t_d	the last placement decision making time
X_h	job observation history
ACT	admission category threshold ($\leq N - 1$)
M_N	decision tree model with N categories

Table 1: Algorithm notation.

- We use a spillover tolerance range, $T_{\text{SPILLOVERTCIO}}$, within which the ACT remains unchanged. Avoiding large adjustments, if $P_{\text{SPILLOVERTCIO}}$ falls below the range lower bound, we decrease the threshold by 1; if $P_{\text{SPILLOVERTCIO}}$ exceeds the upper bound, we increase the ACT by 1.
- This ACT update is triggered only at job arrivals and at fixed decision intervals of t_l seconds, rather than on every job arrival. This reduces the frequency of threshold changes.

When we designed the algorithm, we discovered that considering all the jobs *starting* within the look back window can result in a more accurate estimate of the latest SSD usage information than using all the jobs *overlapping* the look back window. We think this could be the result of jobs with a long lifetime having an outsize effect in such a setting.

5 EVALUATION

We study the following research questions for evaluation:

RQ1: What is our method's performance when integrated into Google's system?

RQ2: What are our method's TCO and TCIO savings?

RQ3: What are our method's TCO savings under different SSD space constraints?

RQ4: How does our method generalize across workloads, or perform with new users and pipelines?

RQ5: Is our ML model practical? Which features contribute most to learning?

As described in Section 1, most of our results are based on a diverse set of workloads written against a shared data processing framework. To show generality, we also demonstrate that the approach works with other workloads. These results, as well as additional experiments and sensitivity analyses, are shown in Appendix C.

5.1 Experimental Setup

Metrics. We evaluate our performance using two metrics: *TCO savings percentage* and TCIO *savings percentage*. As described in Section 3, TCO includes the total expenditure of maintaining a storage systems (such as I/O cost, SSD wearout, etc). TCO savings are relative to the total TCO if all jobs are put on HDD. TCIO measures the actual I/O cost without calculating the SSD wearout and network cost. Given that the SSD wearout cost could differ in different contexts, we believe showing TCIO can help understand the savings purely from an I/O perspective. Similar to TCO savings, we show TCIO savings as the percentage of TCIO savings over the TCIO if all jobs are put on HDD.

Data Collection & Model Training. We collect production traces from *Google* that consist of the historical execution log of the data processing framework and the I/O traces from the distributed storage system. These traces contain jobs' metadata and post-execution measurements, such as TCIO. Section 4.2 explains the features we use. Our training and test dataset each contains one week's data, which are collected from a contiguous two-week time span. We include both read-heavy and write-heavy jobs, reflecting the diversity of workloads across the hyperscale fleet. Block sizes of these workloads vary from several KiB to MiBs.

A key trade-off in our approach is the granularity of model training. We could train one model for every binary, or a joint model that can be used across multiple workloads. To scale to the large number of workloads in our data set, we use the latter approach. Since similar workloads are often run within the same server cluster/data center, we use clusters as the granularity of model training, jointly training one model per cluster and using it for all workloads in this cluster. This has sufficient accuracy (Section 5.5), but nothing precludes us from using a finer or coarser granularity. We use a 15-class gradient-boosted trees model with 300 trees at maximum and a max depth of 6 for all of our models.

End-to-end System Integration. We develop a prototype and deploy it in our production data processing framework and distributed storage system at Google. In the prototype, we first follow Section 4.1 to train the per-cluster model offline. Our trained models are located inside the data processing framework during execution. There is no need to modify user application code. On the compute nodes and inside each job process, the computation framework collects required features and loads the model to perform inference, to generate a categorization result before opening files for writing. The categorization results are passed to the storage cache server, which makes real-time decisions for placement on HDD or SSD. Metrics are collected during the process to evaluate TCO and TCIO.

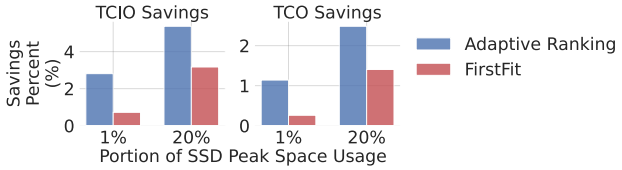


Figure 5: Prototype results.

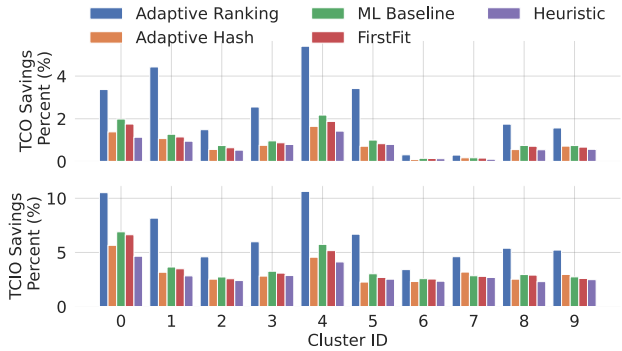


Figure 6: TCO savings (top) and TCIO savings (bottom) from different clusters with fixed SSD quota.

Large-Scale Simulation Setup. We conduct extensive simulation using real production traces at the scale of a cluster. The simulations allow us to perform detailed study of performance and trade-offs at a large scale. Our simulation executes job placement on either SSD or HDD. If a job is placed on SSD but only partially fits, the remaining portion of the job spills over to HDD after filling the available SSD capacity. For experiments varying SSD capacities, we initially set the SSD constraint to infinity to determine the cluster's maximum space usage. We then simulate different scenarios by varying the SSD space quota.

Methods Compared. We compare 7 methods (Section 3): *FirstFit* (simple heuristic), *Heuristic* (advanced heuristic, a modified (Yang et al., 2022)), *ML Baseline* (following (Zhou & Maas, 2021)), *Adaptive Hash* (our method without ML models), *Adaptive Ranking* (our method), *Oracle* TCIO (best theoretical bound when optimizing TCIO), and *Oracle TCO* (best theoretical bound when optimizing TCO).

5.2 Integration in Real Systems [RQ1]

We pick one cluster and implemented two methods in Google's storage system: FirstFit and Adaptive Ranking (ours). A range of data pipelines are selected to generate I/O workloads in this prototype. These pipelines cover a variety of large dataset processing tasks, which generate a wide range of I/O workloads with different intensity and throughput. One category of pipelines is more cost-effective when using HDD, while the other category is more cost-effective to run on SSD. These pipelines are executed continuously in a production cluster, akin to real production pipelines. We set up a dedicated SSD cache so that more precise and disturbance-free results can be measured. A total of 320 worker servers are used to execute the workloads, which includes 16 pipelines and 1024 shuffle jobs in total. The pipelines' combined peak storage usage is 3.6 TiB.

We set up two scenarios where the SSD quota is 1.0% and 20% of the peak theoretical SSD usage limit (Figure 5) respectively, which are common in real-world deployments. For the 1.0% case, our algorithm shows 1.14% TCO savings (**4.38**× over FirstFit). Our method gives 2.48% TCO savings (**1.77**× over FirstFit) in the 20% of the peak workload space usage. The TCIO savings indicate a similar pattern: Adaptive Ranking is 3.90× and 1.69× over FirstFit in the two SSD quota cases respectively.

The end-to-end prototype demonstrates the viability of our design. The measured savings of Adaptive Ranking and improvements over the baseline are in line with the performance in the large-scale simulation studies in Section 5.3 and thus validate our simulation methodology.

5.3 Overall Savings [RQ2, RQ3]

We pick 10 clusters with large TCO to evaluate overall savings. Each of these clusters has thousands of machines. There is considerable variation between workloads within clusters, including video processing, ML, and database queries. Furthermore, the distribution of applications is uneven among clusters. To show performance across different clusters, we fix the SSD quota at 1.0% of the peak SSD space usage and show savings per workload (Figure 6). Our method (Adaptive Ranking) can save over 3.47× at maximum (2.59× on average) compared with the best baselines in terms of TCO savings. The TCIO savings follow a similar pattern. Typically, the TCIO savings increase with SSD quota because SSD cost is not considered. In comparison, the TCO savings initially increase as SSD quota goes up but drop when SSD quota is very large due to high maintenance costs of SSDs. We consider our approach as an effective solution especially when SSD space is limited.

We also evaluate the TCO savings when SSD quota varies. In practice, we want an approach that can adapt to external factors (such as changing SSD quota). Our method consistently saves more TCO than baselines, especially in limited SSD quota cases (Figure 7). The gap between our method (adaptive ranking) and adaptive hash (non-ML) clearly indicates the necessity of our category model. The gap between oracle (best possible in theory) and our method also indicates the remaining headroom for future work.

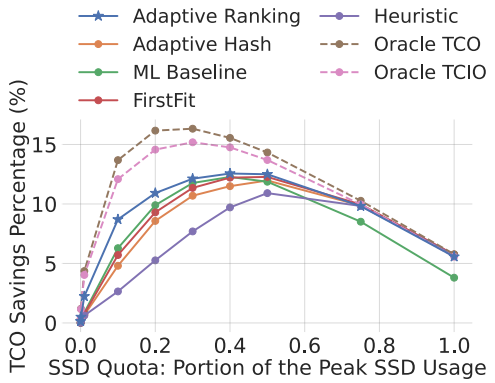


Figure 7: TCO savings.

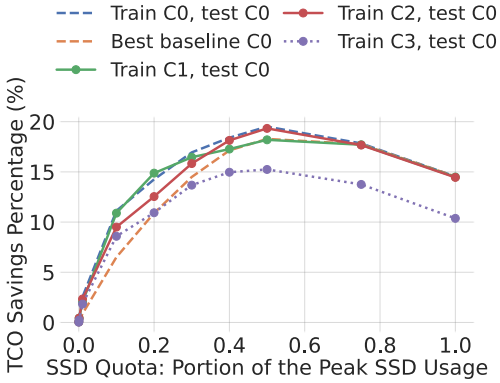
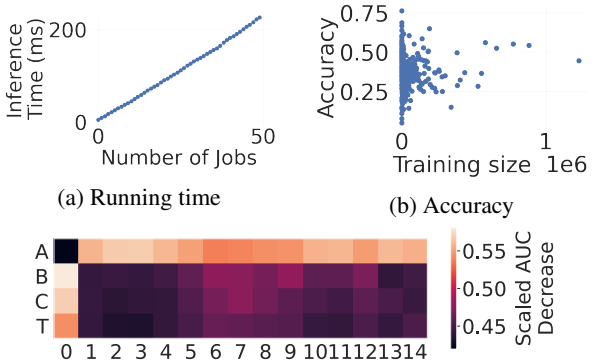


Figure 8: Workload generalization.



(a) Running time

(b) Accuracy

(c) Importance of different feature groups

Figure 9: Model analysis.

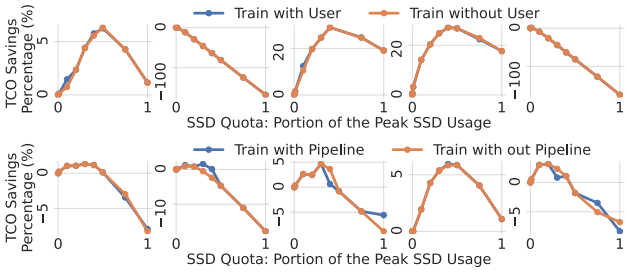


Figure 10: Generalization on new users (upper) and new pipelines (lower). Each figure is for one cluster.

5.4 Generalizability [RQ4]

We explore generalizability of our method across users and workloads. In practice, good generalizability is necessary as infrastructure, user behaviors, workloads, etc. change over time. First, we evaluate the generalizability across workloads (Figure 8). We train one category model for each cluster $C0/C1/C2/C3$, and evaluate their performance on $C0$. $C3$ is a special cluster that only runs certain workloads that are rare in other clusters. We thus find that our method can adapt to unseen workloads, except for outliers.

Second, we evaluate the performance on new incoming users' jobs or jobs from unseen pipelines. We pick the second-largest TCO consuming user and pipeline, which are in different clusters. In evaluation, we compare two training methods: 1) We train the category model with the historical workloads including the user or the pipeline. 2) We train the category model with the historical workloads excluding the user or the pipeline. We evaluate the TCO savings curve under five clusters and show that our method can achieve similar TCO savings on new users or pipelines as in the case where the users or pipelines are in the training set (Figure 10). The blue line (user or pipeline included in the training) and the orange line (user or pipeline excluded in the training) achieve similar savings online.

5.5 Model Analysis [RQ5]

Another practical requirement for ML models is their execution latency and explainability. We show the accumulated inference time of 50 jobs in Figure 9a, where inference takes about 4 ms per job (considerably lower compared to the 99ms of the Transformer model in (Zhou & Maas, 2021)), fast enough for making online placement decisions. Note that our ML model invocation is currently an unoptimized prototype implemented in Python. Potential efficiency improvements could be achieved through further performance optimization, such as using YDF's C++ binding.

We show our model accuracy in Figure 9b. We compare the relationship between model accuracy and the training size across all workloads. The average top-1 accuracy is 0.36 for 15-category classification model and we do not spot a strong correlation between training size and accuracy, which indicates that large data size may not be strictly required.

Next, we analyze how the model makes decisions by assessing feature importance across feature groups, as depicted in Figure 9c. Feature group significance is represented by color

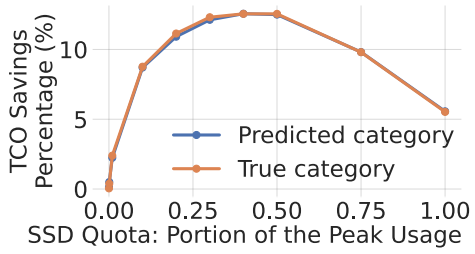


Figure 11: Comparison with using true category.

intensity, with lighter shades indicating higher importance. As discussed in Section 4.2, the features are divided into four groups: A (Historical system metrics), B (Execution metadata), C (Allocated resources), and T (Job timestamp).

To quantify the importance of features in predicting each category, we perform a binary prediction analysis (whether a job belongs to the category or not) for each category. For each feature, we measure the decrease in the area under the ROC curve (AUC) when that feature is excluded from binary prediction tasks. This approach helps us understand how the absence of a feature affects the model's predictive performance. These scores are normalized for comparability within each category. We calculate and present average importance scores for each feature group. In our model, the category 0 is for negative TCO savings and the remaining categories are associated with the ranking of I/O density. Our findings reveal that historical system metrics significantly influence the prediction of I/O density rankings. In contrast, the start time and execution metadata are more critical for predicting whether a job's TCO saving is negative.

We also find that our end-to-end savings may not benefit from more accurate models in Figure 11. Here, we evaluate the performance of the TCO savings assuming we have perfect models for classification prediction. The "Predicted category" is our approach. The "True category" is a method where we replace the category prediction described in Section 4.2 with the ground truth category, representing 100% prediction accuracy.

This observation highlights an important insight: For a given algorithm, there are diminishing returns from further improving model accuracy beyond a certain point. Future improvements in TCO savings are more likely to come from refining the design of the category itself—specifically, its ability to accurately capture job importance—and enhancing its integration with the underlying adaptive algorithm. The synergy between effective category design and the adaptive algorithm plays a crucial role in achieving optimal results.

This finding helps us rethink that the challenges of ML for Systems are not solely about improving model accuracy or learning algorithms. Instead, how the learning problem is formulated and how the ML model is applied within the system are equally important factors in achieving practical performance gains.

6 Related Works

ML in Storage Systems. Prior works have shown the viability of ML for task property prediction in storage systems (Liu et al., 2020; Kaler & Toshniwal, 2023; Singh et al., 2022; Akgun et al., 2023; Vietri et al., 2018; Chakrabortti & Litz, 2020). (Hao et al., 2020) leverages a small neural network to infer SSD performance with fine granularity and help parallel storage applications. (Zhou & Maas, 2021) tackles a related SSD tiering problem with using application-level information and distributed traces by taking inspiration from natural language processing. While the paper focuses on a specific learning problem of mapping textual metadata to storage-related properties, our work focuses on the practical designs and deployment of such models. Additionally, in contrast to existing work, we propose a more granular and practical approach by learning dedicated models for each workload in a cross-layer manner.

Data Placement in Practice. Though ML for systems has been widely explored in different application domains, the state of the art practical solutions for caching or tiering in storage systems are still mostly heuristic (Pakize, 2014; Raj et al., 2012; Zaharia et al., 2009; Downie et al., 2023; Yang et al., 2022; 2023b;a; Eytan et al., 2020; Zhao et al., 2023). Another noteworthy work presents a solver-based solution for task scheduling in the setting where each task contains a list of preferred locations identified prior to scheduling. Their approach formulates the problem as a minimum cost maximum matching problem (Herodotou & Kakoulli, 2021). Although closely related to our work, as discussed in Section 2 and Section 3, the method is not directly feasible in our context, since this approach requires clairvoyant knowledge of jobs' costs and scheduling times.

Appendix D has more discussion of related work.

7 Conclusion

We present a practical approach to data placement in data centers. We identify and solve practical challenges with a cross-layer data placement solution combining application-level ML models with storage-level heuristics. Our approach shows significant TCO savings over the SOTA. We believe that our cross-layer approach presents a methodology for practical ML usage in systems beyond the data placement problem (discussed more in Appendix E).

Acknowledgements: We want to thank Larry Greenfield, Tzu-Wei Yang, Deniz Altınbüken and the anonymous reviewers for invaluable feedback and discussions.

REFERENCES

Abbasloo, S., Yen, C.-Y., and Chao, H. J. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pp. 632–647, 2020.

Akgun, I. U., Aydin, A. S., Burford, A., McNeill, M., Arkhangelskiy, M., and Zadok, E. Improving storage systems using machine learning. *ACM Transactions on Storage*, 19(1):1–30, 2023.

Albrecht, C., Merchant, A., Stokely, M., Waliji, M., Labelle, F., Coehlo, N., Shi, X., and Schrock, C. E. Janus: Optimal flash provisioning for cloud storage workloads. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pp. 91–102, 2013.

Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 143–157, 2011.

Chakraborttii, C. and Litz, H. Improving the accuracy, adaptability, and interpretability of ssd failure prediction models. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 120–133, 2020.

Dong, M., Meng, T., Zarchy, D., Arslan, E., Gilad, Y., Godfrey, B., and Schapira, M. PCC vivace: Online-Learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pp. 343–356, 2018.

Downie, K., Sindam, N., Myers, T., Patterson, J., Gries, W., and Uhse, F. Cloud tiering overview: Cloud tiering heatmap. https://learn.microsoft.com/en-us/azure/storage/file-sync/file-sync-cloud-tiering-overview#cloud-tiering-heatmap, 2023.

Dulloor, S. R., Roy, A., Zhao, Z., Sundaram, N., Satish, N., Sankaran, R., Jackson, J., and Schwan, K. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, pp. 1–16, 2016.

Eytan, O., Harnik, D., Ofer, E., Friedman, R., and Kat, R. It's time to revisit LRU vs. FIFO. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 29–43, 2003.

Guillame-Bert, M., Bruch, S., Stotz, R., and Pfeifer, J. Yggdrasil decision forests: A fast and extensible decision forests library. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, pp. 4068–4077, 2023. doi: 10.1145/3580305.3599933. URL https://doi.org/10.1145/3580305.3599933.

Hao, M., Toksoz, L., Li, N., Halim, E. E., Hoffmann, H., and Gunawi, H. S. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 173–190, 2020.

Hashemi, M., Swersky, K., Smith, J., Ayers, G., Litz, H., Chang, J., Kozyrakis, C., and Ranganathan, P. Learning memory access patterns. In *International Conference on Machine Learning*, pp. 1919–1928. PMLR, 2018.

Herodotou, H. and Kakoulli, E. Trident: task scheduling over tiered storage systems in big data platforms. *Proceedings of the VLDB Endowment*, 14(9):1570–1582, 2021.

Kaler, R. and Toshniwal, D. Deep reinforcement learning based data placement optimization in data center networks. In *2023 IEEE International Conference on Big Data (BigData)*, pp. 2293–2302. IEEE, 2023.

Kanakis, M. E., Khalili, R., and Wang, L. Machine learning for computer systems and networking: A survey. *ACM Computing Surveys*, 55(4):1–36, 2022.

Kang, M. and Park, J. Contragan: Contrastive learning for conditional image generation. *Advances in Neural Information Processing Systems*, 33:21357–21369, 2020.

Kim, H., Seshadri, S., Dickey, C. L., and Chiu, L. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. *ACM Transactions on Storage (TOS)*, 10(4):1–21, 2014.

Klimovic, A., Litz, H., and Kozyrakis, C. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 759–773, 2018.

Kraska, T., Alizadeh, M., Beutel, A., Chi, E. H., Ding, J., Kristo, A., Leclerc, G., Madden, S., Mao, H., and Nathan, V. Sagedb: A learned database system. 2021.

Lee, H., Pham, P., Largman, Y., and Ng, A. Unsupervised feature learning for audio classification using convolutional deep belief networks. *Advances in neural information processing systems*, 22, 2009.

Liu, E., Hashemi, M., Swersky, K., Ranganathan, P., and Ahn, J. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*, pp. 6237–6247. PMLR, 2020.

Maas, M. A taxonomy of ML for systems problems. *IEEE Micro*, 40(5):8–16, 2020.

Mishra, N., Imes, C., Lafferty, J. D., and Hoffmann, H. Caloree: Learning control for predictable latency and low energy. *ACM SIGPLAN Notices*, 53(2):184–198, 2018.

Molla, R. Google, Amazon and Microsoft cloud businesses helped more than double spending on data centers last year. https://www.vox.com/2018/3/15/17124300/google-amazon-microsoft-cloud-200-\percent-jump-data-center-acquisitions, 2018.

Nurmamat kyzy, A., Krettek, A., Altay, A., Goenka, A., Kedin, A., Volpato, B., Chen, C., Dombrova, C., Jayalath, C., McCormick, D., Cavazos, D., Bonaci, D., Halperin, D., Ye, E., Perry, F., Dwivedi, H., Lee, H., Suryawirawan, H., Mejía, I., Malone, J., Anderson, J., Casey, J., Phalip, J., McCluskey, J. R., Sok, K., Knowles, K., Kuligin, L., Liu, M., Gryzykhin, M., Bradshaw, R., Akidau, T., Groh, T., Weise, T., Kirpichov, E., Onofré, J.-B., Iyer, A., Romanenko, A., Estrada, P., Fernández, R., Baetens, M., Rokni, R., Tummalapalli, T., Meiri, U., Zhang, B., Wang, R., Michels, M., Kang, N., Galvan, P., Williams, R., Nanavati, S., Hulette, B., Burke, R., Tymofieiev, V., Pilloud, A., Weaver, K., Oliviera, D., Qiu, R., Zeng, M., Zou, Y., Khanin, A., Kozyrev, I., Kosolapov, A., Hermann, B., Sundhar, S., Herraiz, I., Zhang, Y., Syse, D., Ghorse, R., Hu, Y., Rodriguez Defino, P., Sharma, N., and Uyarer, T. Apache beam: An advanced unified programming model. https://beam.apache.org, 2012.

Pakize, S. R. A comprehensive view of hadoop mapreduce scheduling algorithms. *International Journal of Computer Networks & Communications Security*, 2(9): 308–317, 2014.

Paleyes, A., Urma, R.-G., and Lawrence, N. D. Challenges in deploying machine learning: a survey of case studies. *ACM Computing Surveys*, 55(6):1–29, 2022.

Raj, A., Kaur, K., Dutta, U., Sandeep, V. V., and Rao, S. Enhancement of hadoop clusters with virtualization using the capacity scheduler. In *2012 Third International Conference on Services in Emerging Markets*, pp. 50–57. IEEE, 2012.

Saxena, M., Swift, M. M., and Zhang, Y. Flashtier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 267–280, 2012.

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, 2010. doi: 10.1109/MSST.2010.5496972.

Singh, G., Nadig, R., Park, J., Bera, R., Hajinazar, N., Novo, D., Gómez-Luna, J., Stuijk, S., Corporaal, H., and Mutlu, O. Sibyl: Adaptive and extensible data placement in hybrid storage systems using online reinforcement learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 320–336, 2022.

Vietri, G., Rodriguez, L. V., Martinez, W. A., Lyons, S., Liu, J., Rangaswami, R., Zhao, M., and Narasimhan, G. Driving cache replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

Yang, J., Plasson, N., Gillis, G., Talagala, N., Sundararaman, S., and Wood, R. Hec: improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference*, pp. 1–11, 2013.

Yang, J., Qiu, Z., Zhang, Y., Yue, Y., and Rashmi, K. FIFO can be better than LRU: the power of lazy promotion and quick demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pp. 70–79, 2023a.

Yang, J., Zhang, Y., Qiu, Z., Yue, Y., and Vinayak, R. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 130–149, 2023b.

Yang, T.-W., Pollen, S., Uysal, M., Merchant, A., and Wolfmeister, H. CacheSack: Admission optimization for google datacenter flash caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 1021–1036, 2022.

Zaharia, M., Borthakur, D., Sarma, J. S., Elmeleegy, K., Shenker, S., and Stoica, I. Job scheduling for multi-user mapreduce clusters. Technical report, Technical Report UCB/EECS-2009-55, EECS Department, University of California . . . , 2009.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pp. 15–28, 2012.

Zhao, M., Pan, S., Agarwal, N., Wen, Z., Xu, D., Natarajan, A., Kumar, P., Tijoriwala, R., Asher, K., Wu, H., et al. Tectonic-Shift: A composite storage fabric for Large-Scale ML training. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 433–449, 2023.

Zhou, G. and Maas, M. Learning on distributed traces for data center storage systems. *Proceedings of Machine Learning and Systems*, 3:350–364, 2021.