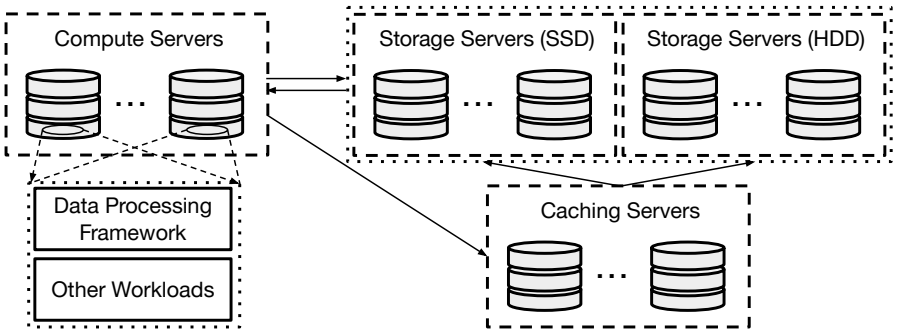

Figure 12: Production Setup Overview

A Production Setup

Figure 12 provides an overview of our distributed production setup. This setup includes three dedicated sets of servers central to storage placement: 1) compute servers, which run data processing frameworks and other workloads; 2) storage servers, which host HDD and SSD devices; and 3) caching servers, which manage SSD tiering decisions.

B Model Features

Our design centers around the intermediate files of data processing frameworks. Section 2.1 introduces the fundamental concepts of how the data processing framework processes input data records. A distributed framework spawns workers to execute tasks. A worker is a process that runs on a server. Workers use shuffling to exchange data between them. 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.

At a higher level, a job comprises three steps. We assume that each worker possesses a number of data records in their working memory. In the first step, each worker writes the data they own into raw intermediate files. Accessing the data in these raw files is inconvenient because they lack a specific order. To address this issue, one or more sorters organize the data records in these files into sorted intermediate files as part of the second step. In the third step, the workers retrieve their required data from the sorted intermediate files back into their working memory, concluding the shuffle job. If feasible, these three steps can be executed concurrently, resulting in temporal overlap.

The I/O density of jobs depends on how these data records are written and read, so we are providing as much internal job-related information from the framework to the model as possible. Internally, the data a workflow needs to process is divided into buckets. A bucket is a unit of work that is assigned to a worker. Each bucket contains a set of tasks that are executed by a single worker. The number of buckets is determined by the data to be shuffled and the number of workers available. Buckets are used to ensure that work is distributed evenly across workers and that no worker is overloaded.

In the first step of a job, the worker shards the data in each bucket into shards, and each shard is assigned to a writer for being written to storage. A writer packs data into stripes and writes one stripe at a time. This enables parallel writing and faster write throughout. The feature we choose, as described in Table 2, reflects how these steps are being executed.

C Additional Results

C.1 Evaluations with non-Data-Processing Framework Workloads

Throughout the paper, we focus on workloads written against the same large-scale data processing framework to evaluate against a wide range of different workloads. We run additional experiments to demonstrate that our prototype and general approach is not limited to this data processing framework, but can handle any workload that supports our distributed storage system.

Note that our "bring your own model" approach means that workloads have a large degree of freedom in terms of producing the predictive category signals that are passed to the storage layer. We demonstrate this flexibility here, by picking diverse workloads that are entirely well-suited for SSD, or entirely well-suited for HDD (i.e., even a model that predicts the same

Features	Feature Group	Description
average_TCIO	Historical system metrics	Average TCIO of the job's historical executions.
average_size	Historical system metrics	Average peak intermediate file size of the job's historical executions.
average_lifetime	Historical system metrics	Average job historical lifetime.
average_I/O density	Historical system metrics	Average I/O density of the job's historical executions.
bucket_sizing_initial_num_stripes	Allocated resources	The initial number of stripes a shard is expected to be divided into. Each stripe contains a couple of data records.
bucket_sizing_num_shards	Allocated resources	The number of shards the working set is expected to be sharded into.
bucket_sizing_num_worker_threads	Allocated resources	Number of worker threads.
bucket_sizing_num_workers	Allocated resources	Number of workers in this job.
initial_num_buckets	Allocated resources	The initial number of buckets the job uses when it was started.
num_buckets	Allocated resources	The number of buckets the current job actually uses.
records_written	Allocated resources	The number of records to be shuffled for a shuffle job.
requested_num_shards	Allocated resources	Number of shards the current working set is requested to be sharded into.
open_time_dayhour	Job timestamp	The hour of the job start time.
open_time_seconds	Job timestamp	The second of the job start time.
open_time_weekday	Job timestamp	The week day of the job start date.
build_targetname	Execution metadata	The target in the build file that is used to build the executable binary.
execution_name	Execution metadata	A user-assigned identifier for the job. Usually set to the binary file name.
pipeline_name	Execution metadata	Name of the pipeline the job belongs to. A pipeline contains multiple jobs.
step_name	Execution metadata	A computer generated step identifier from the workflow's execution graph.
user_name	Execution metadata	Name of the workflow step that is starting the shuffle job.

Table 2: Feature details.

Features	Example Values
build_targetname	//storage/[redacted]/build_manager:[redacted]
execution_name	com.[redacted].[redacted].[redacted].[redacted].trigger2.launcher.Main
pipeline_name	[redacted]_org_[redacted]_indicator_metrics.[redacted]-dims_prod.[redacted].data_importer
step_name	[redacted]-open-shuffle10
user_name	GroupByKey-22

Table 3: Feature examples.

category for each file in this workload would perform reasonably well). We use the adaptive ranking algorithm design to first develop an oracle model based on the workloads' TCO savings and I/O density, then train a model to assign file categories.

Two methods are implemented and compared for the real-world mixed workloads evaluation: FirstFit and our Adaptive Ranking. The real-world evaluation is done using a mix of workloads based on our data processing framework (referred to as "framework workloads") and conventional workloads (referred to as "non-framework workloads"). Our goal is to understand how well these mixed workloads work in the real world. In our evaluation, we maintain a 1:1 framework workloads to non-framework workloads ratio in terms of generated file size footprint.

The following workloads are used for this evaluation:

1. **4 HDD-suitable framework data processing workloads**. These are data processing workloads that perform a small amount of shuffles.
2. **4 SSD-suitable framework data processing workloads**. These are large query workloads that perform a large amount of table joints and therefore need a lot of shuffles.
3. **10 HDD-suitable (low I/O intensity) non-framework workloads**. These are ML training workloads with checkpointing, using the same ML framework that we used for our own models. Since these checkpoint files are kept for longer than a few hours, they are not suitable for being saved to SSD.
4. **10 SSD-suitable (high I/O intensity) non-framework workloads**. These jobs emulate a user workflow that consists of compressing input data, generating (compressed) temporary files, uploading them to a cloud storage, and deleting the temporary files. These workloads generate hot and short lived files.

A total of 320 worker servers are used to execute the workloads. The workloads' combined peak storage usage is 3.8 TiB. All of the four workloads use gradient-boosted tree category models.

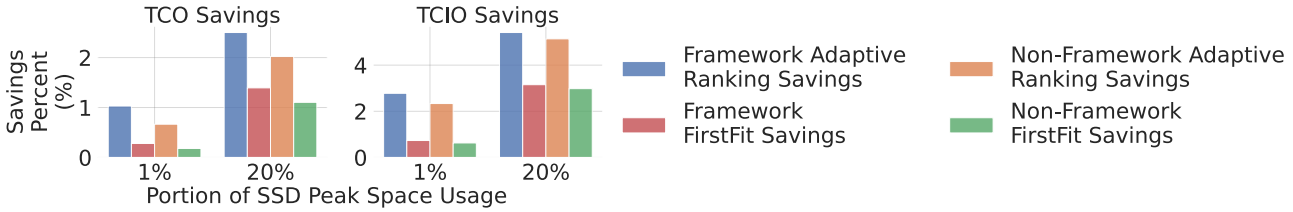

Figure 13: Prototype mixed workload savings.

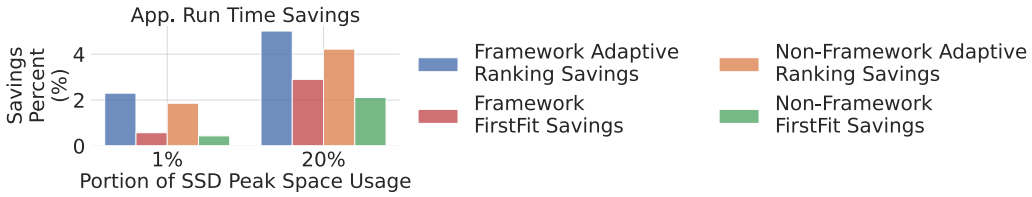

Figure 14: Prototype mixed workload application run time savings.

C.1.1 Storage TCO and TCIO Savings

The measured TCO and TCIO of FirstFit and our Adaptive Ranking are compared with the FirstFit baseline's TCO and TCIO. The results are shown in Figure 13. We see that we get significant TCO and TCIO savings compared to FirstFit, for both our framework and non-framework workloads. This demonstrates that our approach is not limited to workloads written in our data processing framework.

C.1.2 Application-level Performance

We also look into the change of application-level performance introduced by our method. Since our workloads have a fixed amount of work for each execution, we measure the framework and non-framework workload overall execution time as a way to understand the application-level performance. The result is shown in Figure 14. We see that the application-level performance of all workloads improves, in addition to TCO and TCIO savings. Most importantly, no workload shows any regressions. Recall that such savings are expected but *opportunistic* (section 3); i.e., since workloads are written against performance with HDD, our goal is to improve storage costs without degrading application performance relative to this baseline. Any additional performance savings are on top of these goals.

It should also be noted that the application-level performance change depends highly on application's workload composition, most notably the compute to I/O ratio. We select these applications for our evaluation because they are typical in the workloads we need to handle. Other applications' performance change could be very different in these scenarios.

C.2 Sensitivity Analysis

We explore the sensitivity of our method under different hyperparameters below.

Adaptive Algorithm Parameters. We include all combinations of hyperparameters where $T_{\text{SPILLOVERTCIO}} \in \{[0.005, 0.03], [0.01, 0.15], [0.05, 0.25]\}$, look back window time length (seconds) $t_w \in \{600, 900, 1800\}$, and admission decision

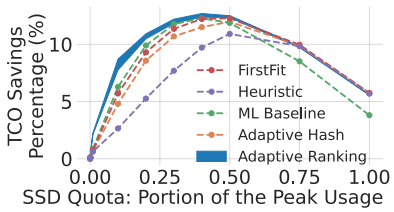

Figure 15: Adaptive algorithm parameters sensitivity.

Method	TCO Savings Percent	Model Top-1 Accuracy
Ours ($N = 2$)	9.25%	73.4%
Ours ($N = 5$)	11.1%	55.6%
Ours ($N = 15$)	12.7%	32.3%
Ours ($N = 25$)	12.6%	24.2%
Ours ($N = 35$)	10.8%	21.2%
Best Baseline	10.7%	/

Table 4: The TCO savings under different category numbers.

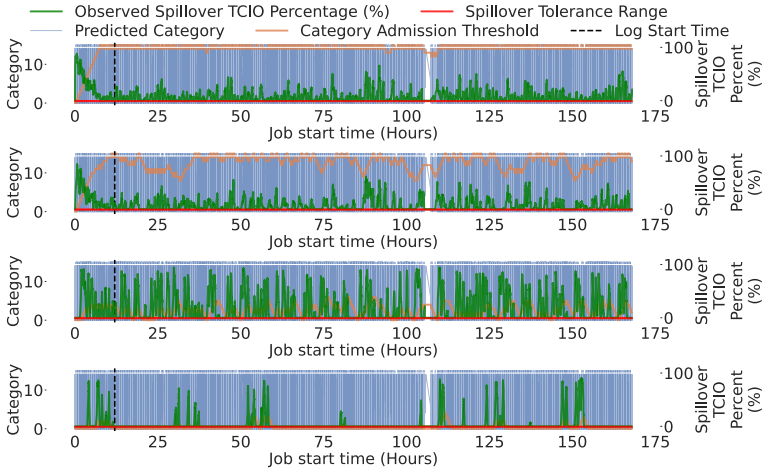

Figure 16: Category change of one workload. From top to bottom, the SSD quota covers 0.01%, 1.0%, 10%, and 50% of the peak SSD space usage under no SSD quota limit. The green line is the observed SPILLOVERTCIO and the orange line represents the category admission threshold. The red area at the bottom of each figure is $T_{\text{SPILLOVERTCIO}}$.

effective time $t_l \in \{600, 900, 1800\}$. We evaluate the sensitivity of the TCO savings for the same set of workloads in fig. 7. For each parameter combination, we apply the same parameter settings to all the workloads in the group. In fig. 15, the blue area in the figure presents the upper bound and lower bound of TCO savings under different SSD capacities across different hyperparameter combinations. Our solution is not sensitive in terms of hyperparameter selection in the adaptive algorithm.

Sensitivity on Category Numbers. Our evaluation utilizes the 0.1 SSD portion setting with all the algorithm parameters maintain consistent. It is critical to select an appropriately large number of categories to enable the model to effectively distinguish the cost across jobs without increasing the model's capacity for fine-grained category prediction. We present the impact of category numbers N on end-to-end TCO savings in table 4. A model with smaller category number achieves higher accuracy but fails to optimize the end-to-end TCO savings due to its limited granularity. Conversely, increasing the number of categories enhances granularity but at the cost of accuracy, diminishing the TCO savings.

C.3 Adaptive Category Selection Dynamics

To demonstrate the dynamics of our adaptive algorithm, we present the pattern of category threshold change and spill over percentage in fig. 16. We track the threshold change for 1 week online. Our adaptive category selection algorithm can adjust the category admission threshold to a higher range when SSD quota is limited and allow more category admissions when SSD space is plentiful.

D DETAILED RELATED WORKS DISCUSSION

Prior works have shown the viability of machine learning for task property prediction in storage systems. (Hao et al., 2020) leverages a small neural network to infer SSD performance with fine granularity and help parallel storage applications. The method learns a binary latency model and pre-calculate an inflection point for each model during a labelling stage. The key benefit is model simplicity and fine granularity of prediction, enabling more complicated applications online within latency requirements. (Zhou & Maas, 2021) tackles a problem related to our setting in data placement with methods that leverage

application-level information and distributed traces in a way inspired by ideas 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.

Other applications of machine learning in storage systems include training one monolithic model for the entire storage system (not deployable in warehouse-scale setting due to adaptability): applying imitation learning for cache replacement to approximate an optimal oracle policy (Liu et al., 2020), guiding the placement algorithm model through reinforcement learning (Kaler & Toshniwal, 2023; Singh et al., 2022); predicting properties in other aspects of data placement: improving a storage system through optimizing readahead and NFS read-size values with machine learning models (Akgun et al., 2023), utilizing ML to improve on existing cache replacement strategies (LRU, LFU, etc.) (Vietri et al., 2018), and predicting future task failures through ML (Chakraborttii & Litz, 2020).

Multiple machine learning techniques have also been proposed in broader system problems (Kanakis et al., 2022; Maas, 2020), ranging from resource allocation (Mishra et al., 2018), memory access prediction (Hashemi et al., 2018), offline storage configuration recommendation (Klimovic et al., 2018), database query optimization (Kraska et al., 2021), to networking applications (Dong et al., 2018; Abbasloo et al., 2020). Although the nature of these applications is different from data placement in storage systems, they all show evidence that machine learning can be used in systems and benefits from domain-specific formulations.

Data Placement in Practice. Though machine learning 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.

Hadoop offers three caching schedulers: FIFO (Pakize, 2014), Capacity (Raj et al., 2012), Fair (Zaharia et al., 2009). Spark supports FIFO, Fair. For each user, Azure tracks the last-accessed files and make the placement based of the self-tracked access history (Downie et al., 2023). (Yang et al., 2022) presents a novel adaptive cache admission solutions for Google, of which we implement a modified version in our comparison.

Very recent works have also started rethinking the best practical solution within the heuristic-based domain. (Yang et al., 2023b;a) consider a modified FIFO for cache eviction, which achieves good scalability with high throughput on production traces from Twitter and MSR. (Eytan et al., 2020) revisits the effectivenss of LRU versus FIFO and finds that FIFO exhibits better overall cost than LRU on production traces, including IBM COS traces. (Zhao et al., 2023) proposes new heuristics for storage, specifically tailored for machine learning workloads at Meta.

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 the Section 2 and Section 3, the method is not directly feasible in our context. The primary challenge in adopting such a solver-based approach in our setting lies in the lack of jobs' cost at scheduling time.

E DISCUSSION

Our "bring your own model" idea can be adopted in other deployments and frameworks. The solution flow of our system is: 1) Setting an optimization objective (TCO savings in our case). 2) Designing a 'hint' (workload model output) passed from each workload from the application layer to the caching layer. Our hint is job importance in terms of TCO savings. 3) Picking jobs of different categories based on feedback from system utilization.

This BYOM flow remains adaptable to other applications. The key components that vary are the objective function, available model features, and how system utilization is quantified.

While Table 2 lists specific features used in our model, these features fit into four general categories: historical information, job start time, execution information, and allocated resource information. The general categorization remains applicable though the detailed features may vary, allowing for adaptation across different organizations.