HEIMDALL: Optimizing Storage I/O Admission with Extensive Machine Learning Pipeline

Daniar H. Kurniawan
University of Chicago
Chicago, IL, USA
MangoBoost Inc.
Bellevue, WA, USA
daniar.h.k@gmail.com

Rani Ayu Putri
Bandung Institute of Technology
Bandung, West Java, Indonesia
University of Chicago
Chicago, IL, USA
raniayu@uchicago.edu

Peiran Qin
University of Chicago
Chicago, IL, USA
peiranqin@uchicago.edu

Kahfi S. Zulkifli
Bandung Institute of Technology
Bandung, West Java, Indonesia
sbhnkahfi@gmail.com

Ray A. O. Sinurat
University of Chicago
Chicago, IL, USA
rayandrew@uchicago.edu

Janki Bhimani
Florida International University
Miami, FL, USA
jbhimani@fiu.edu

Sandeep Madireddy
Argonne National Laboratory
Chicago, IL, USA
smadireddy@anl.gov

Achmad Imam Kistijantoro
Bandung Institute of Technology
Bandung, West Java, Indonesia
imam@itb.ac.id

Haryadi S. Gunawi
University of Chicago
Chicago, IL, USA
haryadi@cs.uchicago.edu

Abstract

*This paper introduces HEIMDALL, a highly accurate and efficient machine learning-powered I/O admission policy for flash storage, designed to operate in a black-box manner. We make domain-specific innovations in various ML stages by introducing accurate period-based labeling, 3-stage noise filtering, in-depth feature engineering, and fine-grained tuning, which together improve the decision accuracy from 67% up to 93%. We perform various deployment optimizations to reach a sub-*µs *inference latency and a small, 28KB, memory overhead. With 500 unbiased random experiments derived from production traces, we show HEIMDALL delivers 15-35% lower average I/O latency compared to the state of the art and up to 2× faster to a baseline. HEIMDALL is ready for user-level, in-kernel, and distributed deployments.*

CCS Concepts: • **Software and its engineering** → **Operating systems**; • **Computing methodologies** → **Machine learning approaches**.

Keywords: ML for systems; I/O admission control; File and storage systems; Operating systems; Distributed systems

ACM Reference Format:
Daniar H. Kurniawan, Rani Ayu Putri, Peiran Qin, Kahfi S. Zulkifli, Ray A. O. Sinurat, Janki Bhimani, Sandeep Madireddy, Achmad Imam Kistijantoro, and Haryadi S. Gunawi. 2025. HEIMDALL: Optimizing Storage I/O Admission with Extensive Machine Learning Pipeline. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3717496

1 Introduction

Several recent advances in data centers [5, 7, 63] have been directly driven by the wider adoption of flash arrays. SSDs, with their microsecond-level access speeds, have become the preferred choice in the storage hardware stack. However, a piece of dark cloud above the high performance of SSDs is their *non-deterministic* internal operations [31, 32, 41, 51, 81], such as garbage collection (GC), internal buffer flushing, and wear leveling. These growing background complexities cause significant tail latency amplification, leading to unpredictable interruptions that degrade user experience. For example, GC operations can negatively affect performance by increasing latency by up to 60× [51]. To address this challenge, I/O admission control (referred to as replica selection in some literature) has emerged to avoid submitting I/O requests to flash arrays undergoing intensive internal operations.

There is a vast body of research on I/O admission control techniques that rely on heuristics, such as hedging [34], replica scoring, and rate limiting [33, 36–38, 74, 87], forecasting and rate controlling [53, 86]. However, with the growth of storage industry, recent real-world workloads show a more complex pattern due to the increasing hardware capability, enabling more work to be done. As a result, heuristics-based

EuroSys '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/2025/03
https://doi.org/10.1145/3689031.3717496

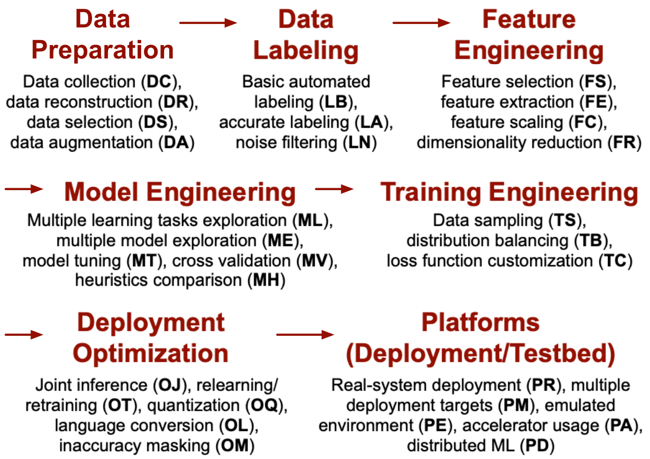
Data Preparation
Data collection (DC), data reconstruction (DR), data selection (DS), data augmentation (DA)
Data Labeling
Basic automated labeling (LB), accurate labeling (LA), noise filtering (LN)
Feature Engineering
Feature selection (FS), feature extraction (FE), feature scaling (FC), dimensionality reduction (FR)
Model Engineering
Multiple learning tasks exploration (ML), multiple model exploration (ME), model tuning (MT), cross validation (MV), heuristics comparison (MH)
Training Engineering
Data sampling (TS), distribution balancing (TB), loss function customization (TC)
Deployment Optimization
Joint inference (OJ), relearning/ retraining (OT), quantization (OQ), language conversion (OL), inaccuracy masking (OM)
Platforms (Deployment/Testbed)
Real-system deployment (PR), multiple deployment targets (PM), emulated environment (PE), accelerator usage (PA), distributed ML (PD)

Figure 1. Machine learning pipeline (§1).

I/O admission control falls short due to its predefined rules that impede its adaptability. To address these limitations, recent research has seen a shift toward leveraging machine learning (ML) across various areas of storage due to the learning ability that overcomes the rigidity of heuristic-based algorithms. More and more challenges of complex decision-making in various contexts shift to ML-powered, including I/O admission [26, 32, 80], caching [43, 46, 68, 77, 82], configuration tuning [17], deduplication [64], failure detection [10, 22, 56], indexing [21, 54, 73], prefetching [9, 72], scheduling [55], and many others.

While ML-based solutions often outperform traditional heuristics, many studies fail to explore the full scope of the ML pipeline during the design of these systems. These *gaps* occur when key stages of the ML design process, as shown in **Figure 1**, are insufficiently addressed. Skipping or oversimplifying these stages can hinder the model's potential in terms of accuracy, performance, and adaptability. This is particularly evident in the case of ML-powered I/O admission control systems. When tested on modern devices and recent I/O traces (from MSR Cambridge, Alibaba, and Tencent [1, 61, 83]), several ML-powered I/O admission controls [26, 32, 47] showed a significant drop in accuracy—averaging 67%, far below their original claims. In the context of I/O admission control, low accuracy is particularly problematic, as false admits and reroutes can negatively impact I/O latency. These findings underscore the importance of a more thorough execution of the ML pipeline to achieve better-performing models.

In response, Heimdall introduces a more robust ML-powered I/O admission control system, carefully developed by following the key stages of the ML pipeline. By infusing with deep storage domain knowledge and rigorously optimizing the model at every stage, Heimdall is designed to achieve higher accuracy and better performance, addressing the challenges of modern storage workloads in latency-critical environments. Heimdall operates as a black-box system that handles I/O requests, making admission or redirection decisions while monitoring request latency to detect the impact of SSD internal processes. Heimdall introduces three technical contributions.

First, Heimdall introduces the notion of "period-based labeling", derived from our observation of production traces. We observed that SSD internal management process doesn't only affect a single I/O but consecutive I/Os in a period, hence called period-based labeling. This data labeling technique enables us to teach a more accurate ML model. This technique can also be applied in other research areas related to SSDs and scheduling.

Second, to ensure Heimdall is suitable for real-world production systems, we introduce a learning granularity mechanism that is ideal for scenarios where coarse-grained decisions are sufficient. By allowing coarse-grained predictions, Heimdall trades a small accuracy loss for a significant boost in inference throughput. This approach generalizes well to a common challenge in ML-for-systems: the difficulty of real-world deployment. To the best of our knowledge, we are the first to apply this technique to optimize I/O admission control. We further validate Heimdall's real-world applicability by successfully integrating it into both the Linux kernel and Ceph RADOS.

Third, we present a generic and extensible ML pipeline that enables the development of Heimdall's admission control system. This flexible pipeline highlights the importance of incorporating domain knowledge and can be adapted to address a wide range of storage system challenges.

Finally, we evaluate our solution on production traces from companies (MSR, Alibaba, and Tencent) with three levels of integration: user-level storage (for fast and large-scale evaluation), Linux kernel (for mimicking real deployments), and Ceph RADOS (for distributed storage settings). Our comprehensive evaluation with 500 experiments shows that Heimdall delivers 15-35% lower average latency compared to popular algorithms, such as hedging and advanced admission heuristic and ML models [32, 74], and up to 2× faster to a baseline. Moreover, we achieve sub-µs inference latency, up to 0.08µs on a 2.30GHz processor.

We conclude by demonstrating how Heimdall 's optimized I/O admission system can serve as a foundation for further research and innovation in ML-driven storage solutions. While developed for I/O admission control, Heimdall's extensive ML pipeline can also support broader explorations in storage optimization by enabling students and researchers to experiment with new techniques within the pipeline or extend its application to other storage challenges.

2 Background and Motivation

Admission problem: The admission problem is fundamental for operations such as job submission [11, 20, 40], VM placement [69], cached data [80], RPCs [84], and I/O requests [30, 32, 45]. The admission policy needs to decide

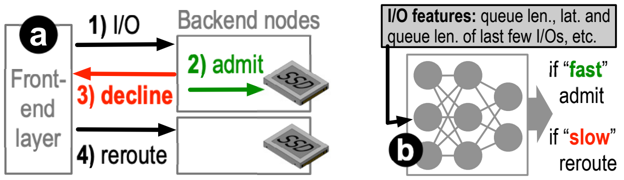

Figure 2. I/O admission (§2). *(a) I/O admission decision and (b) neural-network-based decision in every backend node.*

whether to admit the operation to an underlying resource, delay, or reroute it to another resource. Such a policy is useful for resources that exhibit tail-latency behavior where most of the time the operations are fast but sometimes (*e.g.*, 1-10% of the time) are slow because of resource contention.

I/O admission: The I/O admission problem encompasses several different research objectives, such as reducing excessive flash writes [80] and minimizing tail latency [32]. We focus specifically on I/O admission at the block level in parallel, redundant flash storage arrays that use data replication, with the goal of reducing tail latency. For example, in data centers, redundancy mechanisms such as RAID ensure fault tolerance by storing replicated data, but they also introduce opportunities for performance optimization. Prior work, such as FusionRAID [35], Tiny-Tail Flash [81], and EC-Cache [66], has demonstrated that reconstructing late data from the full stripe can often be faster than waiting for a slow I/O response. These findings suggest that redundancy-aware scheduling can be leveraged to optimize storage performance. I/O admission works by selecting which replica an I/O request should be submitted to in order to escalate the I/O request latency. As illustrated in **Figure 2a**: (1) The front-end layer sends an I/O request to a backend SSD storing the data. Each backend node makes admission decision to (2) *admit* the request to the underlying SSD or to (3) decline the request and ask the front-end layer to (4) *reroute* it to another node that has the replica.

Flash storage: Admission decision is useful to reroute I/Os from flash devices that are experiencing heavy resource contention from GC, internal buffer flush, wear leveling, and not to mention, bursty workloads. Without proper admission/rerouting, all of these can induce read tail latencies and increase the average latency. Since write tail latencies are very rare due to device-internal write buffers [31, 32, 52], we focus on optimizing read latencies.

ML-based policies: I/O admission can be based on ML models such as LinnOS [26, 32], which uses a light neural network to predict contention inside black-box SSDs. As shown in **Figure 2b**, LinnOS is designed to make admission decisions at a uniform *per-page* granularity (4KB I/O). Consequently, it does not include *I/O size* as an input feature. Instead, LinnOS bases its decisions on historical latency and I/O queue length features, such as the latencies of the last few I/Os and the queue lengths at their arrival times. Using this information, the model predicts whether an incoming I/O will be "*fast*" (hence, admit the I/O) or "*slow*" (hence, reroute the I/O). LinnOS is deployed at the kernel block layer of every flash device.

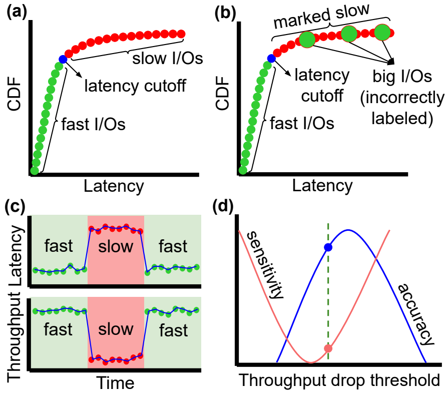

Figure 3. Accurate labeling (§3.1). *In all figures, a dot represents an I/O.* ***(a)*** *Fast / slow cutoff, latency based.* ***(b)*** *Inaccurate labeling, latency CDF with annotated big I/Os.* ***(c)*** *Timeline figure, period-based labeling.* ***(d)*** *Gradient descent.*

Training: To make accurate *binary* "fast/slow" predictions, an ML model like LinnOS must be trained first. Before enabling admission decision, a storage operator can log the characteristics of the last 15 minutes I/Os, recording their static/runtime features and the I/O latencies. The operator then labels every logged I/O as "fast" or "slow" based on some labeling algorithm (more in Section 3.1). During training, the model learns which I/O patterns result in slow I/Os for the workload-device pair. The neuron weights from training are then applied to the in-kernel model for deployment.

Accuracy: The ML model can make two inaccurate decisions: *false admits*, when the I/O is predicted to be "fast," but apparently experiences slowness, or *false reroutes*, when it predicts "slow," but there is no busyness. Our recent evaluation found the accuracy of LinnOS' 4-year-old model degraded to 67% as it failed to keep up with modern workloads and faster SSDs.

3 HEIMDALL's Pipeline

We begin by describing our solution motivated by the above-mentioned challenges. Throughout this section, we will show how we significantly improve the I/O admission control accuracy by leveraging each step in the design process.[1] In particular, we make domain-specific innovations in the data analysis stage with accurate labeling (§3.1) and noise filtering (§3.2); the modeling stage with thorough feature engineering (§3.3), model exploration (§3.4), and hyperparameter tuning (§3.5); and the training stage (§3.6).

3.1 Accurate Labeling

We first evaluated the automated labeling that prior works performed [19, 27, 28, 32, 65] in the context of latency-based

[1] Throughout this section, we use ROC-AUC [23] for accuracy, but later in Section 6.4, we report various accuracy metrics.

```
1. Function AccurateLabeling ()
2.    Input  : Data D {size, throughput, latency}
3.    Output: Data D {size, throughput, latency, label}
4.    high_lat, low_thpt = CalcThreshold(D)
5.    MAX_DROP = CalcThptDropThreshold(D)
6.    thpt_median = CalcMedian(throughputs)
7.    for io in D do:              // Initialization
8.       io.label = 0; io.mark_start = 0
9.       if IsBusy(io, high_lat, low_thpt, MAX_DROP) do:
10.         io.label = 1       // Start of the TailZone
11.   for io in D do:
12.      if io.label == 1 do:    // Label the TailZone
13.         while io.next.throughput < thpt_median do:
14.            io.label = 1    // 1 = decline; 0 = admit
15.            io = io.next
```

Figure 4. Accurate labeling algorithm (§3.1).

modeling. We found that several methods use *latency-based* algorithms to decide the *latency cutoff*, as illustrated in **Figure 3a**, where the algorithm labels the I/Os in the training data set with "fast" or "slow" based on the inflection point (cutoff) generated by the algorithm.

While latency-cutoff algorithms work well in certain domains—for example, in networking [65] where per-packet/per-request size remains stable, or in storage domain where the per-page latency model can only make inferences on per-4KB I/O [32]—this method does not work for ML models that make decision at the whole, variable I/O level ranging from one-page (4KB) to big request (2MB). For example, in **Figure 3b**, a *large* I/O (the red dot) is labeled as "slow" here because its measured latency in the dataset is larger than the cutoff. However, this is *inaccurate* because even if the I/O is rerouted to another device, the I/O will *still* be "slow" due to its large size.

To rectify the problem, we found that *period-based* algorithms give the best result for our problem domain. That is, instead of deciding which specific I/Os should be marked slow or fast, we label based on periods (window of time) where our algorithm guesses whether the device is in fast period (*e.g.*, no internal contention) or slow period (*e.g.*, experience GC contention, etc.). Although we cannot exactly predict the occurrence, we can still discover some patterns in the data to narrow down the plausible tail latency segments in the dataset. For example, in **Figure 3c**, all the I/Os in the slow period (*e.g.*, where latency spikes and throughput drops happen) will be labeled as "slow."

Our algorithm is composed of 3 stages, as shown in **Figure 4**. **(a)** First, in line 9, we categorize the relationship between latency and throughput. We should only be suspicious of device busyness when latency is high and throughput is low at the same time. We observed that internal contention causes throughput drops and latency spikes. Throughput is more sensitive for detecting the start and the end of such events since throughput also takes I/O size into account. **(b)** Thus, we use latency and throughput thresholds (declared in line 4) to decide when latency looks high or throughput looks low.

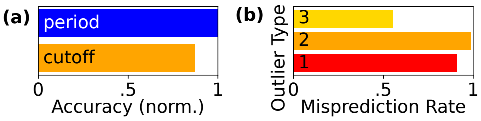

Figure 5. The importance of preprocessing (§3.1 §3.2). *(a) Cutoff vs period-based labeling. (b) Noise misprediction rate.*

Finding threshold values that work across different devices and workload characteristics is *challenging*. We use a gradient descent-based method to pick the proper thresholds across various SSD and workload characteristics to *balance sensitivity and accuracy*. For example, **Figure 3d** shows a global optimum for accuracy and sensitivity, shown as the blue and red lines, respectively. Finally, **(c)** based on these values, we decide when the busy period starts and ends (lines 12 to 15). **Figure 5a** shows the accuracy improvement we obtain by migrating from cutoff-based labeling to a more accurate period-based labeling, emphasizing the labeled data's better learnability. In the evaluation (§6.4), we demonstrate that accurate labeling improves Heimdall's accuracy by 5.5%, resulting in accuracy as high as 93%.

3.2 3-Stage Noise Filtering

To minimize the impact of noisy training data on the model, we introduce a domain-specific, 3-stage noise filtering process. This process targets (1) outliers within slow period, (2) outliers within fast period, and (3) short noises. Our outlier removal specifically targets noise arising from irregular or non-representative events that do not meaningfully contribute to tail latency, hence not conflicting with the goal of improving tail performance. As shown in **Figure 5b**, these outliers often lead to model mispredictions, making them disruptive rather than informative. Aligning with the evaluation findings in **Figure 14a** in Section §6.4, we find that the three types of noise cumulatively degrade accuracy by 16%. Eliminating them allows the model to focus on learning and addressing prolonged contention patterns more effectively, ultimately improving tail performance rather than diminishing it.

In the first stage, we remove *outliers within the slow period*, as illustrated in **Figure 6a**. Here, long sequences of slow I/Os may indicate the device's internal busyness. However, sometimes a few I/Os can be "lucky" and hit the internal device cache even though the device is busy with other activities that only affect NAND-level reads and writes. Thus, as shown in the figure, we remove these "fast" outliers, specifically I/Os that have lower latency and higher throughput than the respective median values within the period.

In the second stage as shown in **Figure 6c** and **Figure 6d**, we remove *outliers within the fast period*, the opposite of the first case above. These slow I/Os could happen due to some other rare device idiosyncrasies such as read retries due to voltage mismatch [15, 60, 88], error check and correction

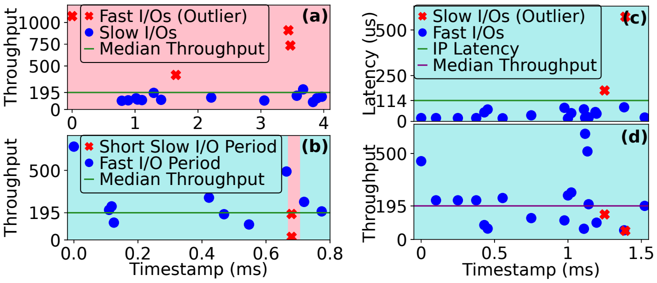

Figure 6. Noise filtering (§3.2). *Sample data from Alibaba trace.* ***(a)*** *Outliers within slow period.* ***(b)*** *Short noises. Outliers within fast period:* ***(c)*** *latency and* ***(d)*** *throughput.*

(ECC) [60], and many others. Since these rare cases are transient errors in nature, removing them from the dataset will increase the model's accuracy.

The data now becomes "cleaner," but we still find a slight irregularity as a result of our labeling process. We observed a *short burst of a slow period* (*e.g.*, only 3 consecutive I/Os), which is unlikely caused by internal device contention, as illustrated in **Figure 6b**. Because supplying such short bursts could confuse the model, in the third stage of the filtering, we employ the same gradient-descent method as in **Figure 3d** in Section 3.1 to find a reasonable threshold that will provide high accuracy but low sensitivity. In most datasets, we find a quick burst of 3 (or less) consecutive "`slow`" I/Os should be removed. Overall, our 3-stage noise filtering improves accuracy further by 16% on average (§6.4).

3.3 In-Depth Feature Engineering

Originally, typical request-based traces contain basic features such as request arrival time, size, and I/O type (read/write). Deriving more features from the original ones will help ML model to grab more characteristics, increasing the accuracy at cost of higher computation overhead. Since we work on latency-sensitive system, we need to balance the obtained model accuracy and the computation overhead by extracting enough advanced features, selecting the best set of important features, and scaling the numbers well to avoid uneven weighting of different features.

To ensure a fair evaluation of feature engineering (*e.g.* feature extraction, feature selection, etc.), we use a neural network (NN) model, as it avoids architectural constraints found in tree-based models, such as depth limitations in decision trees. This flexibility makes NN particularly suitable for identifying variations in feature quality without being restricted by rigid structural assumptions. While prior work [32] have already performed basic **feature extraction** by including latency and I/O queue length observed when every I/O is submitted, **feature selection** by removing timestamp and disk ID, and **feature scaling** by using digitization, they end up with having 31 features. As a result, each inference costs 2ms overhead. We seek to answer the question: *"Without sacrificing the accuracy, can we use simpler model with less number of features to reduce the computation overhead?"*

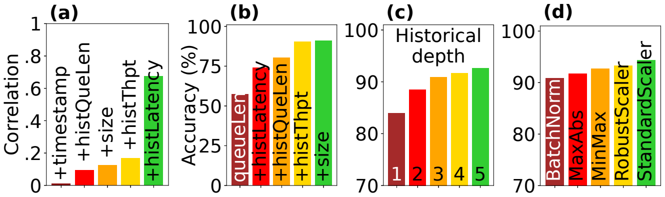

Figure 7. Feature engineering (§3.3). ***(a)*** *Correlation values of each feature,* ***(b)*** *accuracy improvements attributed to each feature,* ***(c)*** *model accuracy on various historical depths, and* ***(d)*** *model accuracy on different normalization methods.*

First, we do **feature selection** by removing features with low correlation scores, such as I/O arrival time (timestamp), which indicates that there is little to no correlation towards whether the I/O induced tail latency behavior. **Figure 7a** ranks the features by their correlation to the decision.

Next, we conducted **feature analysis** on the resulting features. **Figure 7b** shows how the features affect the overall model accuracy, confirming earlier finding that the five main features (the queue length, historical queue length, historical latency, historical throughput, and I/O size) are crucial in improving the accuracy.

Third, we also varied the historical depth (N) of certain input features to determine the amount of past information needed by the model to obtain reasonable accuracy. By historical depth, we refer to the information of the last N I/Os (such as the most recent queue lengths and I/O latencies). This information can imply, for example, if most recently, we observe an I/O with high latency but a short queue length, the device is busy internally. **Figure 7c** shows that N=3 is sufficient to improve accuracy across various datasets.

Finally, we explored **feature scaling** techniques to reduce model bias to specific features while maintaining low computation overhead. We tried various normalization methods, as summarized in the first three bars of **Figure 7d** [8, 85], and found that min-max gives the best accuracy on average. Standardization methods, such as robust and standard scalers, deliver higher accuracy but are not feasible for our domain because of their high memory overhead for keeping all the historical latency values for standard deviation and quantile calculations. Min-max normalization, in contrast, requires only the minimum and maximum values, making it both accurate and lightweight.

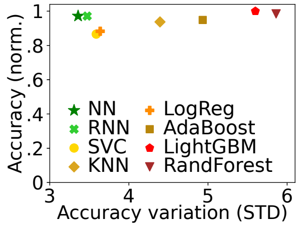

Figure 8. Model exploration (§3.4). *NN model achieves high and stable accuracy.*

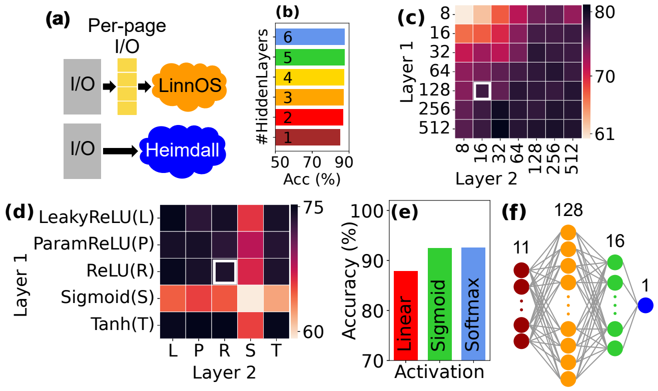

Figure 9. Hyperparameter tuning (§3.5). *(a) Per-page vs. per-I/O, (b) hidden layers, (c) number of layers, (d) activation functions, (e) output layer, and (f) final NN design.*

3.4 Model Exploration

With the chosen feature set, we perform **multiple model explorations** to identify the most suitable model for our problem domain. To ensure fairness, each model undergoes light hyperparameter tuning until no significant accuracy improvements are observed. **Figure 8** summarizes our findings, where the x-axis is the accuracy variation (measured across datasets) and the y-axis is the normalized accuracy; the upper left of the figure is a more suitable model. Overall, we found that the neural network ("**NN**") achieves good accuracy with the highest stability compared to other models.

3.5 Neural Network Hyperparameter Tuning

We conduct **hyperparameter tuning** to determine the optimal number of layers, neurons, and activation functions to balance the accuracy and complexity. As a result of disciplined feature and model engineering, our new NN depicted in **Figure 9f** is significantly simpler than LinnOS' NN model due to these reasons:

(a) First, as depicted in **Figure 9a**, since LinnOS uses a cutoff-based per-page labeling, its model can only make inferences on every 4KB request. Therefore, a big I/O must be split into small uniform-sized block I/Os, increasing the number of inference for each I/O. Meanwhile, as we use period-based per-I/O labeling, we only need one inference for each I/O of any size.

(b) Second, compared to LinnOS which only uses one hidden layer, we use 2 hidden layers. **Figure 9b** shows that the most impactful accuracy growth comes from adding the 2nd hidden layer.

(c) Third, to minimize the computation overhead, we use 128 and 16 neurons for the first and second hidden layers, respectively (while LinnOS uses 256 neurons in just one layer). In **Figure 9c**, the x- and y-axis represent the number of layers in the 1st and 2nd hidden layers, respectively, and the cell color represents the accuracy achieved. We select the lightest model design which gives relatively high accuracy (darker color).

(d) Fourth, we kept ReLU for the activation function of the hidden layers. The x- and y-axis in **Figure 9d** represent the permutation of the 1st and 2nd layer's activation functions, respectively. We chose ReLU due to its resulting high accuracy (darker color), light overhead, and simple computation compared to others.

(e) Finally, for the output layer, we experimented with softmax, linear, and sigmoid [70, 85]. Based on the results shown in **Figure 9e**, we opted for a single-neuron sigmoid. This differs from LinnOS' 2-neuron output layer which bears the consequence of doubling the computation when propagating the gradient from its neighboring hidden layer.

3.6 Training

Due to the inherent nature of tail latency, there exists data distribution imbalance between slow and fast I/Os, where the fast I/O dominates the overall latency distribution. To tackle this problem within training process, we tried **biased training** by customizing the weighted loss function [76] to penalize the model when admitting the slow I/Os. However, we see insignificant improvement or even *worse* results. Upon analysis, different datasets resulted in different optimum loss values due to the variations of slow and fast I/O distribution, thus not feasible.

Other possible methods are data sampling and data selection. Since oversampling and undersampling might expose some risk [75], we tackle this problem by making sure our data selection process (§3.3) includes some periods with heavy write I/Os (to trigger device background activities) and further augment the data (rerate and resize).

4 Deployment Optimizations

After discussing modeling and training, we now shift to deployment, focusing on our efforts to optimize inference latency through joint inference techniques and Python-to-C conversion and optimization.

4.1 Negligible Inference Latency

Many storage systems, such as the Linux block layer and Ceph [78], are written in C, not Python. This makes it difficult to use optimized inference libraries like TensorFlow or PyTorch, which operate in user space. Embedding these libraries into latency-critical systems, such as the Linux kernel, would introduce significant overhead. Therefore, to improve inference latency in real deployments, we must convert our models from **Python-to-C**. To achieve this, we follow a three-step process: Python-to-C/C++ conversion, gcc optimization, and quantization, allowing us to reduce inference time to *sub*-µs levels.

First, with a careful and manual *Python-to-C++ conversion*, we reduce the inference time to 20µs. Second, we use additional gcc *optimization* to reduce the execution time, avoiding quick compilation that sacrifices the performance.

Components		**Integration + Eval**	
Dataset preparation	2.5 K	In User level	3.7 K
Design pipeline	3.6 K	In Linux kernel	2.1 K
Optimizations	1.2 K	In Ceph RADOS	2.3 K
Retraining	0.2 K	Evaluation module	5.3 K
Total: 20.9 K			

Table 1. Implementation scale (§5). *HEIMDALL is written in 20.9K lines of code (LOC) mainly in Python and C/C++.*

We opted for -O3 since it gives the highest optimizations while still obeying the strict compliance to language standards and retaining the computation precision, which speeds up the inference to 0.08µs.

Finally, we perform *quantization* to reduce the computational complexity of our model and minimize the memory footprint. We multiply the weights by 1,024 for all layers and quantize the bias of each layer correspondingly to match the scale. We use 1,024 because we can capture the non-zero digits from most of the weights within 4 decimal points. The final latency drops to 0.05µs per inference.

We also ran tests on various CPUs and found that interestingly the inference latency can vary by an order of magnitude. For example, we get 0.12µs latency on AMD Ryzen 9 5900HS 3.30GHz and 0.08µs latency on AMD EPYC 7352 2.30GHz. On Apple M1 Pro 3.20GHz, it is even faster, at 0.05µs. We leave further investigation for future work.

4.2 Joint/Group Inference

In some deployments, making admission decision on every I/O might be too fine-grained, leading to high CPU overhead under intensive I/O workload and limited resources. Recent approaches, such as LAKE [26], propose GPU batching, which only works well in large-batch inference by exploiting parallelism. While improving throughput, the host-to-GPU latency overhead incurs additional delays.

In contrast, we are inspired by **joint/group inference** [14, 29, 59]. We modify the model to be able to take features of up to P I/Os. Joint inference is more efficient than batching, where it makes *one* inference on behalf of all of the P I/Os (imagine a green traffic light for P cars to pass through), while batching still requires the model to be run P times and make P decisions.

In HEIMDALL, one can specify the model's granularity for inference, ranging from 1 I/O per-inference up to P I/Os per-inference. For all inference granularity, HEIMDALL uses the same model architecture. However, the number of historical data (see §3.3) are perpendicular to the granularity (P). Storage operators can decide to set the granularity. The challenge of developing joint inference models lies in the feature selection phase since we do not want to increase the model complexity by aggregating all input features from the P I/Os. Instead, we believe that the most up-to-date device behavior is reflected in the most recent I/Os. Thus the prioritization of features from the most recent I/Os, ignoring most features from the rest of the I/Os. For instance, for P=5, we still only supply the historical information of the last three I/Os before this group of 5 I/Os (§3.3), keeping the model light from the reduced redundancy. Later in Section 6.5, we evaluate the tradeoffs, *e.g.*, higher P leads to higher throughput/performance but only reduces accuracy slightly.

5 Implementation Scale

Our effort to build a playground of applying machine learning for storage system is shown in **Table 1**, which breaks down our 20.9 KLOC implementation of HEIMDALL pipeline. The left column shows the main components, which include dataset preparation scripts, all the design stages (§3), deployment optimizations (§4), and retraining (§7), and the right column shows our three levels of integration in user-level storage (§6.1), Linux kernel (§6.2), and Ceph RADOS (§6.3), including our evaluation module that contains re-implementation of other policies. This extensible playground can be easily reused for other research.

6 Evaluation

Our comprehensive evaluation is set up as follows: • **Data size:** We use 2 TB of raw I/O block traces from Alibaba, Microsoft, and Tencent [1, 61, 83] and generate 11 TB of intermediate data for all the experiments. • **Target deployments:** We integrate HEIMDALL into *user-level storage* (for fast, large-scale evaluation), *Linux kernel* (to mimic real deployments), and *Ceph* (for distributed evaluation). • **Machine and SSDs:** By default, we use Chameleon's Storage-NVMe node which has AMD EPYC 7352 2.30GHz 24-Core CPU with 256 GB DRAM. We use 10 different SSD models from various manufacturers[2] • **Train/test:** All the experiments use 50:50 train-test methodology which provides a more balanced representation compared to the 80:20 split, ensuring the evaluation set remains completely unseen during the training process, guaranteeing that our evaluation metrics are unbiased and accurately reflect the model's performance on unseen data.

This section will address the following key questions:

- How does HEIMDALL's performance compare to state-of-the-art algorithms in large-scale evaluations? (**§6.1**)
- How does HEIMDALL perform when deployed in Linux kernel? (**§6.2**)
- How does HEIMDALL scale to improve I/O latency in a multi-node Ceph cluster? (**§6.3**)
- How does each step in the ML pipeline contribute to HEIMDALL's overall performance? (**§6.4**)
- How can HEIMDALL's inference throughput be optimized with joint inference? (**§6.5**)

[2] Intel (DC-S3610 and DC-P4600), Samsung (850-PRO, 970-PRO, PM961, PM1733, PM1725a, MZV-PV128HDGM, and MZH-PV128HDGM), and Hitachi SN260.

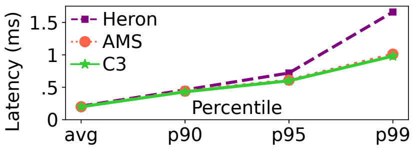

Figure 10. Heuristics-based algorithms comparison (§6.1). *C3 outperforms other state-of-the-art algorithms.*

- What are the CPU and memory overheads, as well as the training time, for Heimdall's light neural network model? (**§6.6** and **§6.7**)

6.1 Large-Scale Evaluation

To evaluate Heimdall comprehensively and **unbiasedly**, we conducted a large-scale evaluation with **hundreds of random time windows ("traces")** from various real-world traces from Alibaba, Microsoft, and Tencent [1, 61, 83]. To ensure that these hundreds of traces represent various workload characteristics, we picked the traces based on five criteria which are read/write ratio, size, IOPS, randomness, and overall ranking. For each criterion, we picked time windows with p10, p25, p50, p75, p90, and p100 values, with respect to all the time windows in the long, multi-day traces.

To further increase the variability of our datasets, we also apply 5 different data augmentation functions (0.1× rerate, 0.5× rerate, 2× rerate, 2× resize, and 4× resize). This approach simulates even more demanding scenarios than typically encountered in real-world applications. From this large dataset pool, we **randomly** picked **500 traces**. Each trace is then capped at 3 minutes long, which contains between 100k to 10 millions of I/Os. A 3-minute trace is long enough for the underlying devices to exhibit some tail latencies due to GC, write amplification, and other contentions, but at the same time, it is short enough for a large-scale experiment.

To ensure a realistic evaluation, we focus on a light-heavy workload combination, reflecting real-world scenarios with varying load conditions that can lead to tail latency if I/O selection is not handled properly. We categorize a trace as light if the I/O count is less than 300k. In this combination, it is essential to avoid blindly rerouting I/Os from the heavy trace to the light one. Doing so could inadvertently overload the other device, resulting in reduced overall performance. An efficient model should only decline and reroute I/Os when absolutely necessary.

Consistent with the trace source, our traces do not include thread IDs. Instead, we follow standard practice by executing concurrent I/O operations through deploying multiple threads ($N\geq8$) on the client side. These threads submit I/O streams in parallel respecting the I/O's timestamp, following the behavior of concurrent applications issuing I/O operations. Additionally, we conducted comprehensive tests across consumer-level and enterprise-grade SSDs, which are prone to garbage collection (GC) under heavy load.

For faster experiment setups (just for this subsection), we deploy Heimdall in a user-level I/O replayer to simply mimic application-level storage with direct I/Os. For each experiment, we simulate a 2-way replicated storage environment using a machine equipped with 2 Samsung SSD 970 PRO 1TB SSDs. During each experiment, we run a random trace where the I/Os will pass through a Heimdall model specific to the device, which determines the I/O admission decision. If an I/O is declined, it is redirected to the other device, which will be admitted by default.

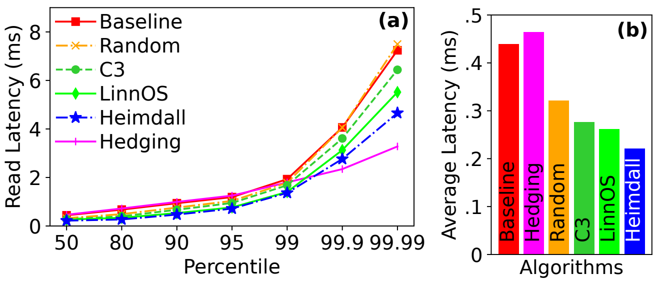

Figure 11. Large-scale evaluation (§6.1). *Subfigures **(a)** and **(b)** depict the read latency at percentiles ranging from p50 to p99.99 and the average latency.*

To properly evaluate Heimdall, we compare it against several well-known policies, including a simple always-admit approach with no rerouting (baseline), random target selection (i.e., sending I/O to a randomly chosen device), and state-of-the-art admission and rerouting algorithms such as **C3** [74], **AMS** [36], **Heron** [33], **LinnOS** [32], and **hedging** [24]. In the first experiment, we focus on selecting a representative algorithm from the heuristic-based category, as many proposed algorithms fall into this group. This selection simplifies the subsequent evaluation while still capturing the best performance of algorithm group. As shown in **Figure 10**, we evaluated three key heuristic-based algorithms: AMS, C3, and Heron. The result shows that C3 and AMS deliver very similar performance, with both providing lower latency compared to Heron. Given C3's wide adoption in the industry and its competitive performance, we selected C3 as a representative for heuristic-based algorithms and evaluate it further in the next experiment, as well as in the Kernel (later in §6.2).

Building on this, **Figure 11a** highlights the tail latencies across various percentiles, further illustrating Heimdall's superiority over state-of-the-art algorithms. These latencies are the *average* percentile from the 500 experiments, hence showing *Heimdall wins in large-scale, unbiased experiments.* Furthermore, **Figure 11b** shows another impressive outcome of Heimdall in delivering the *lowest average latency.*

We can also see that while hedging delivers shorter tail latencies above p99, its average latency is far worse than Heimdall. For instance, hedging at p98 (between 0.75ms–1.5ms), after a 2ms timeout, a backup I/O is sent, causing too much overload (instability) which leads to *higher* average latency than the baseline. Therefore, hedging appears ineffective for low-latency requests.

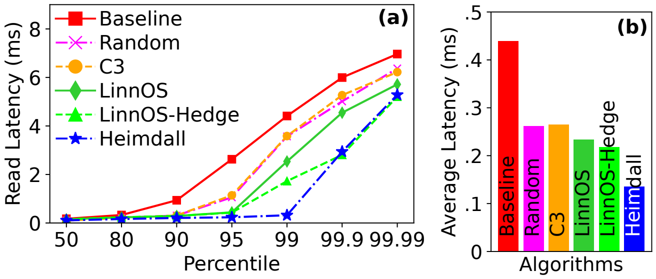

Figure 12. Kernel-level evaluation (§6.2). *Heimdall achieves* ***(a)*** *most stable latencies at p50 to p99.99 percentiles and* ***(b)*** *lowest average latency.*

6.2 Kernel-Level Evaluation

While the previous section shows a user-level deployment, this section briefly shows Heimdall results when deployed inside the Linux kernel block layer. As we already did a large-set experiment previously with homogenous datacenter Samsung SSD 970 PRO 1TB SSDs, here we provide a different setup by running a portion of Microsoft trace on a machine with two consumer-grade SSDs, Intel DC-S3610 and Samsung PM961. As demonstrated in **Figure 12a**, Heimdall also works effectively for in-kernel deployment and on heterogenous SSDs, outperforming other methods by delivering faster latencies (in the y-axis) at various percentiles (in the x-axis). **Figure 12b** further shows a *successful in-kernel deployment of Heimdall*, delivering the lowest average latency, 38-48% faster compared to the non-baseline methods.

6.3 Wide-Scale Evaluation

So far, we have analyzed Heimdall on a single machine with multiple drives. This section shows Heimdall’s performance with a “wide-scale” setup by deploying Heimdall in Ceph distributed storage system [78]. For this, we use 10 Chameleon Ice Lake machines, each with two 2.30 GHz 40-core Intel(R) Xeon(R) Platinum 8380 with 256 GB DRAM. On each machine, we deploy two Ceph Object Storage Daemons (OSDs) to set up a replicated storage with primary and secondary OSDs. Due to limited availability of dual SSD machines in the Chameleon cluster, we set up the OSDs on top of FEMU-emulated SSDs (100 GB each) [50]. To send requests to these 20 OSDs, we create 20 client nodes and run noise injectors to see how the admission policies react to noisy neighbors. Based on the results from the prior section, we compare three methods: baseline, random, and Heimdall. In the baseline Ceph setup, each request is directed to the primary OSD, whereas in the random setup, requests are randomly load-balanced. LinnOS is excluded from this evaluation because it only supports fixed 4KB per-page predictions, making it incompatible with Ceph’s variable-sized I/O operations. Since Ceph workloads involve diverse request sizes across a distributed storage environment, a system like LinnOS cannot generalize effectively because it is designed for uniform, page-level decision-making. This further highlights Heimdall’s flexibility, as it seamlessly integrates with Ceph and other large-scale systems, making it easier to deploy and adapt to real-world storage environments.

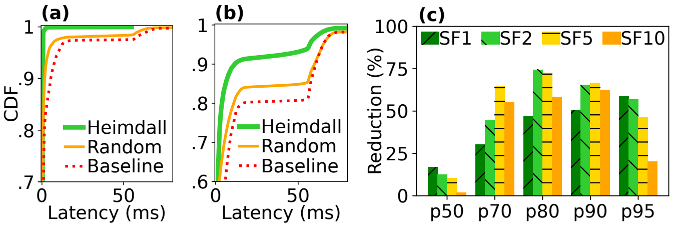

Figure 13. Wide-scale evaluation (§6.3). *CDF latency on Ceph cluster for* ***(a)*** *SF = 1,* ***(b)*** *SF = 10;* ***(c)*** *latency reduction at percentiles ranging from p50 to p95 across various SFs.*

The latency CDF in **Figure 13a** shows that *in a wide-scale evaluation, Heimdall also delivers the best result.* Furthermore, we also vary the “scaling factor” (*SF*). According to Google’s seminal “Tail at Scale” paper [24], an end-user request can consist of multiple parallel “sub-requests” to different destination servers, but the end-user request is only considered complete when all the sub-requests are complete. To show Heimdall’s benefits when the tail is amplified by scale, we vary the *SF* factor (*e.g.*, *SF* = 10 means an end-user request has 10 parallel sub-requests. **Figure 13b** shows the latency CDF when *SF* = 10, showing that tail behavior starts appearing in baseline at p75 and Heimdall can efficiently cut the large portion of the tail area. Furthermore, **Figure 13c** shows the tail latency reduction of Heimdall vs. random (in the y-axis) at various percentiles (x-axis) across a variety of scaling factors (as shown in the legend). Heimdall wins in all the scenarios (positive percentage of latency reduction).

6.4 Accuracy

Behind Heimdall’s strong performance lies the high accuracy it achieves. *This section dissects how every design decision contributes to increasing Heimdall’s overall accuracy.* There are five main metrics of accuracy that we use: ROC-AUC, PR-AUC, F1-Score, FNR, and FPR, based on the number of true/false positives and negatives. Their equations can be found here [23]. In our case, *true positive (TP)* implies that the model correctly identifies the I/O as “slow” and *false positive (FP)* implies the opposite (marked as “slow” but the I/O will actually be fast if submitted to the device).

In our domain, ROC-AUC is the preferred metric for imbalanced datasets as it balances sensitivity and specificity [13]. More specifically, the tail latencies are the minority compared to the fast latencies. Unlike `accuracy` or `precision`, ROC-AUC provides a comprehensive evaluation that considers the trade-off between true positive and false positive rates at various classification thresholds.

As shown in **Figure 14a**, we measure the resulting ROC-AUC score (x-axis) of every step-by-step optimization (y-axis). These optimizations contribute to enhancing both the dataset and the model which increases HEIMDALL’s overall

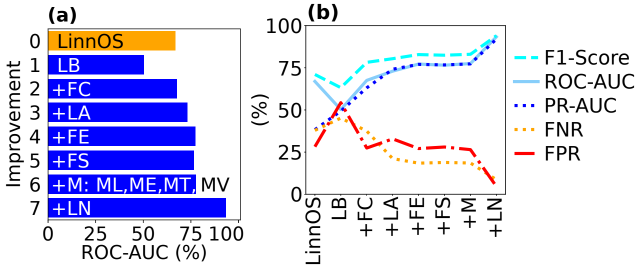

Figure 14. Accuracy evaluation (§6.4). *Contribution of each step; comparing* ***(a)*** *ROC-AUC only and* ***(b)*** *all metrics of LinnOS towards Heimdall design steps: Basic Labeling (LB), Feature Scaling (FC), Accurate Labeling (LA), Feature Extraction (FE), Feature Selection (FS), Model Engineering (M), and Noise Filtering (LN).*

accuracy at every step of the way. The numbering, (*n*), below corresponds to the y-axis values in Figure 14a.

(0) We first measured LinnOS' accuracy as a baseline, 67% average across over 100 random datasets. In measuring LinnOS' accuracy, to ensure fair comparison, we used the LinnOS architecture and applied the same setup and conditions as for our Heimdall architecture during testing. LinnOS' accuracy dropped significantly compared to what the authors reported in the paper four years ago. The model is outdated given the different behavior of modern SSDs (faster latencies) and more recent workloads released by the community, which highlights the necessity to re-design the model with a more extensive ML pipeline.

(1) Here, we began developing Heimdall using LinnOS' original feature set, model architecture, and *basic cutoff-based labeling (LB)*. However, we removed digitization—the feature scaling technique used in LinnOS—because it is specifically designed for *per-page* (4KB) I/O admission and assumes uniformly sized I/Os. Since Heimdall's main approach moves away from per-page decisions to handling variable-sized I/Os, retaining digitization would introduce bias and distort the learning process. Removing it ensures a more meaningful comparison, allowing us to directly evaluate the impact of period-based labeling in a broader I/O admission scenario. This adjustment led to a 16.8% drop in accuracy, bringing it down to 50.2%. However, this drop is valuable as it establishes a controlled lower bound, providing a clear baseline to quantify the improvements introduced by period-based labeling and subsequent optimizations. By shifting our focus to variable-sized I/Os in this stage, we create a more flexible and realistic evaluation framework that better reflects modern storage workloads.

(2) We then applied *min-max scaling (FC)* (§3.3) to normalize the input features instead of using digitization (as in LinnOS), resulting in a slightly better accuracy (67.5%) compared to LinnOS.

(3) Afterward, we crafted our more *accurate period-based labeling (LA)* method (§3.1) and used it to replace LinnOS' cutoff-based labeling. As a result, we increase the accuracy by 5.5%, bringing it to 73%.

(4) Then, with additional *feature extraction (FE)* (§3.3), including I/O size and historical throughput, the accuracy gains another 4% improvement (now reaching 77%) as Heimdall now can discern patterns related to the volume and rate of data being transferred.

(5) To reduce the model's inference overhead, we applied a *feature selection (FS)* method (§3.3) that maintains accuracy without inducing degradation.

(6) Likewise, we employed *model engineering (M)* (§3.4 and §3.5) steps consisting of learning task exploration (ML), model exploration (ME), hyperparameter tuning (MT), and validation (MV), to find a balance between the model's complexity and accuracy. We obtained a minimalist model capable of maintaining the same level of accuracy achieved thus far.

(7) Finally, our *noise filtering (LN)* (§3.2) proves to be one of the most important contributions, which increases the accuracy by 16%, leading to 93% model accuracy.

Furthermore, to show that we are not biased over one accuracy metric, **Figure 14b** shows the resulting accuracy (in the y-axis) across the five accuracy metrics (the five lines) as we add the step-by-step contributions (in the x-axis). Overall, as more optimizations are introduced, ROC-AUC, PR-AUC, and F1-Score continue to increase. Furthermore, the false-negative and false-positive rates (FNR and FPR) also continue to decrease as desired.

6.5 Joint/Group Inference

We now discuss Heimdall's joint inference performance (§4.2). As shown in **Figure 15a**, with the default setting of Heimdall (without joint inference, shown by joint size = 1), it can only receive 0.5 mIOPS workload before its latency spikes to 2µs. However, with a joint size = 9, Heimdall maintains latency under 2µs even with a 4 mIOPS workload on 1 CPU core, an 8× heavier workload than the default Heimdall. Note that this latency includes queueing delay, rendering it slower than our fastest inference latency (§4.1).

Joint inference reduces accuracy, as quantified in **Figure 15b**. For example, transitioning from the default Heimdall to 9 I/O joint inference, the accuracy drops from 88% to 81% in the median value. The figure shows the resulting accuracy distribution across 50 random datasets. Given the results above, we believe that joint size = 3 is appropriate for balancing out the throughput/accuracy tradeoff. When deploying Heimdall, storage administrators can also adjust the joint size dynamically.

We further evaluate the effectiveness of joint inference by comparing it with LAKE [26]. LAKE enhances LinnOS'

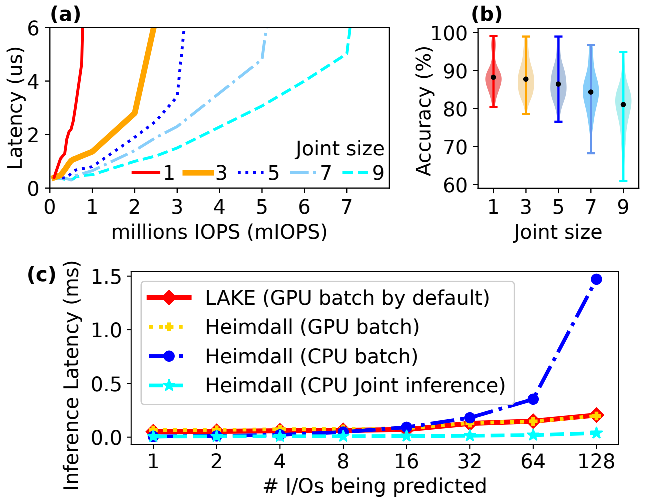

Figure 15. Joint inference (§6.5). *(a) Throughput stability, (b) model's accuracy, and (c) comparison with LAKE [26].*

inference throughput by providing easier GPU accessibility in kernel space, enabling batched ML inferences to be offloaded to the GPU. For a better comparison, we implement three versions of Heimdall: GPU batch (accelerated by LAKE's mechanism), CPU batch, and CPU-based joint inference. **Figure 15c** presents the results on the GeForce GTX 1660 SUPER GPU, where we vary the number of I/Os to be inferred simultaneously from 1 to 128. First, LAKE and Heimdall perform similarly under GPU accelerations, where Heimdall gains an upper hand up to 0.01ms. Second, compared to GPU-based approaches (LAKE and Heimdall GPU batching), Heimdall CPU-based joint inference reduces the inference latency by up to 10×. Third, compared to Heimdall CPU batching, Heimdall CPU-based joint inference has much lower inference latency when the number of simultaneous I/Os scales up. This is because batching has the computational complexity of N×, where N is the batch size. However, joint inference shares similar input features among multiple I/Os and merges them into a single inference, making the computation overhead negligible for larger batch size. Thus, Heimdall with joint inference can tolerate more I/Os intensive scenarios and provide better accessibility without relying on GPU.

6.6 CPU and Memory Overhead

We now assess Heimdall memory and CPU overhead. **Figure 16** shows that, compared to LinnOS, our model achieves **(a)** 2.4× less memory overhead, 68KB vs. 28KB, and **(b)** 2.5× less CPU overhead. Heimdall has a total of 3472 weights and biases, which is smaller than LinnOS' total of 8706 weights and biases. Furthermore, our model has 2.4× fewer multiplication operations, 3472 vs. LinnOS 8448 multiplications. Moreover, Heimdall makes significantly fewer inference decisions since it operates on a per-I/O basis instead of per-page decisions like in LinnOS. To further reduce the overhead, Heimdall can be deployed with joint size = 3 (represented by Heimdall-J3, light-blue bar), resulting in 85% less CPU overhead compared to LinnOS.

6.7 Training Time

Finally, we report the average training time of Heimdall, which depends on the number of I/Os. For every 1 million I/Os we use for training, it takes us 16.8 seconds of preprocessing time on CPU and 3.7 seconds of training time on GPU. Preprocessing includes labeling, extracting features, normalizing, and shuffling the data. We trained the model once on each trace file, assuming that each trace belongs to different hardware and workload, thus requiring a learning phase before usage. The next question to answer is how much data to train on and how should we retrain the model in a long deployment scenario. This significant research question falls outside the scope of this paper. Answering it would necessitate further exploration of the ML pipeline, as we will delve into in the next section.

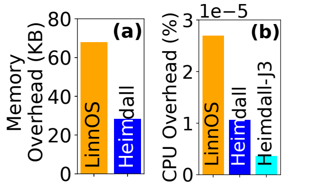

Figure 16. Overhead (§6.6). *(a) Memory & (b) CPU.*

7 Retraining for Longer Deployment

In reality, ML models are deployed for the long term. In this context, we conduct a *preliminary long-term evaluation of Heimdall by testing it on an 8-hour real-world trace with one of the most "challenging" traces where accuracy fluctuates in the long run.* Here, we used a Tencent trace where the write IOPS is 2× more than the read IOPS, triggering more GC activities. Furthermore, this trace exhibits an almost constant I/O interarrival time, causing all devices to experience similar workloads and heavy utilization simultaneously.

Figure 17a shows Heimdall's accuracy after a single training session using the initial 1, 5, and 15 minutes of the trace. We measure the accuracy within a 10-minute window (a dot is the model's average accuracy in the last 10 minutes). We can reach two simple conclusions. First, a longer training trace (*e.g.*, a 15-minute trace) results in better long-term overall accuracy, but requires longer training time (§6.7). Second, accuracy fluctuates over time, with a min-max accuracy of 63%–82%. This is also known as *model's performance drift*, which could stem from factors like shifts in workload behavior (input drift), device/environment changes (concept drift), and others [79].

To address this, we built a preliminary *retraining policy* that monitors the model's accuracy every minute and triggers retraining when the accuracy drops below 80%. To keep retraining light, we only retrain using the last 1 minute before the trigger. **Figure 17b** shows the result, where the vertical blue lines represent the times when retraining is triggered. More specifically, within this 8-hour window, retraining occurs 37×, each utilizing an average of 816k I/Os

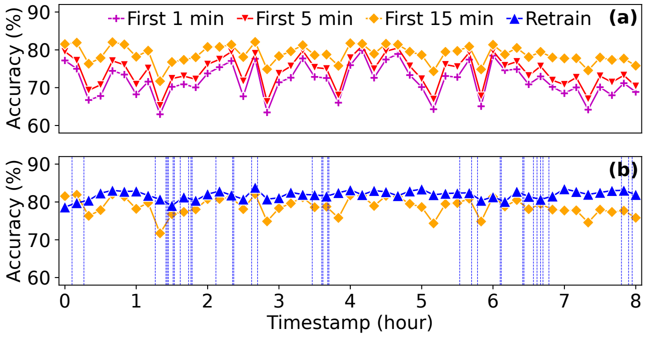

Figure 17. Long-term evaluation (§7). *Heimdall performance with different training methods: "First N min" trains the model once using only the first N minutes of the workload, while "Retrain" uses a simple retraining strategy described in §7. Vertical blue lines mark the time when retraining was triggered.*

ML Methods →	DDDD	LLL	FFFF	MMMMM	TTT	OOOOO	PPPPP
	CRSA	BAN	SECR	LETVH	SBC	JTQLM	RMEAD
LinnOS [32]	xxxx	x..	xx..	..x.x	..x	..xxx	x....
LAKE [26]	xxxx	x..	xx..	..x.x	..x	..xxx	x..x.
Baleen [80]	xx..	x..	xx..	xxx.x	xx.	.....	x.xx.
Cacheus [68]	xxx.	x..	.x..	..x.x	...	.....	..x..
GLCache [82]	xx..	x..	xx..	..x.x	x..	xx.x.	x....
XStore [77]	xx.x	x..	.x..	.x..x	x..	.x.xx	x...x
Bourbon [21]	xxx.	x..	.xx.	xx..x	x..	...xx	x....
LeaFTL [73]	xx..	x..	.x..	..x.x	...	...xx	x.x..
Rolex [54]	xxx.	x..	.x..	x	...	.x.x.	x...x
KML [9]	xxx.	x..	xxx.	.xxxx	x..	..xx.	x....
Voyager [72]	xx..	xx.	xx..	..x.x	...	..xxx	..x..
DeepSketch [64]	xx..	x..	.x.x	xxxx.	.x.	...x.	..xx.
OSML [55]	xx..	x..	.xx.	xxxxx	..x	...x.	xx.x.
Llama [57]	xx..	x..	xx..	x.x.x	x..	...xx	x....
TraceRNN [12]	xx..	x..	xx..	x.x..	x.x	.....	x....
Heimdall	xxxx	xxx	xxx.	xxxxx	.x.	xxxx.	xxx..

Table 2. Usage of ML methods in ML-for-storage literature (§8) . *The table shows the usage of ML methods in ML-for-storage papers. Each column has a two-letter acronym that represents an ML method shown earlier in Figure 1. For example, "$\overset{D}{C}$" in the first column denotes "Data Collection." "**X**" implies **use** of the method and "**.**" implies **absence/no-use**.*

which can be completed in a couple of seconds (§6.7). *This initial result also points to more research questions.* For example, the presence of consecutive retraining instances (vertical blue lines that are close to each other) suggests that some retraining decisions may not be useful. Second, we cannot expect the last 1-minute trace before the retraining session is available because per-request logging is turned off by default due to the significant overhead [16].

Overall, these findings suggest that *there are more topics to explore in this long machine learning pipeline for storage* (as in **Figure 1**) such as efficient retraining, continual learning models [48, 49, 58], and model management [39, 44, 67, 71]. There are many research questions to ask. When to retrain the model? How to avoid catastrophic forgetting during retraining? How to detect performance drifts? What are the I/O characteristics that can provide hints of workload drifts? How often and how much data to use to check for drifts?

8 Discussions and Future Works

With Heimdall's success in achieving high accuracy and reducing tail latency while maintaining sub-µs overhead in real-world deployments like Linux kernel and Ceph storage systems, we believe that its achievements will ignite discussions and raise important questions for future research:

8.1 Heimdall's Broader Impact

Heimdall is the result of applying various machine learning methods in a disciplined and meticulous manner, enriched with deep domain knowledge. Incorporating this domain expertise into the design of the ML-based solution proved to be *crucial.* As a result, Heimdall achieved a significant performance improvement compared to previous approaches to the same problem. Therefore, despite the surge in ML applications for storage domains, we believe that ML's full potential has yet to be unlocked.

To evaluate to what extent this longer ML pipeline is applied to ML-for-storage research, we summarized recent research papers in **Table 2**. The five-row groups in the table represent papers in popular storage domains, such as admission [26, 32, 80], caching [68, 77, 82], indexing [21, 54, 73], prefetching [9, 72], and miscellaneous categories [12, 55, 57, 64]. We conclude that the ML-for-storage literature has *gaps* in exploring various stages of this pipeline.

For example, in the data labeling stage, prior works only employ one labeling method, hence opening an opportunity to increase the accuracy by integrating domain knowledge to a more "accurate labeling" and possibly "noise filtering" to avoid "garbage in, garbage out" [18]. In the feature engineering stage, "feature scaling" is not fully studied, potentially causing the model to assign disproportionate importance to certain features over others [62]. Finally, there is a big chance of applying our "joint/group inferences" in order to increase efficiency and make it feasible for production system deployment. Many also did not consider the need for "retraining/relearning" for long-term deployment, potentially in "multiple layers/targets." Thus, there exists a big opportunity in escalating the performance and reliability of ML-powered applications.

As highlighted in the last row of Table 2, Heimdall's pipeline covers more machine learning approaches. This more rigorous methodology attributes to Heimdall optimum performance shown through the evaluation section (§6). Applying each method we devised in this paper to all papers in Table 2 is out of our scope and will be left as future work.

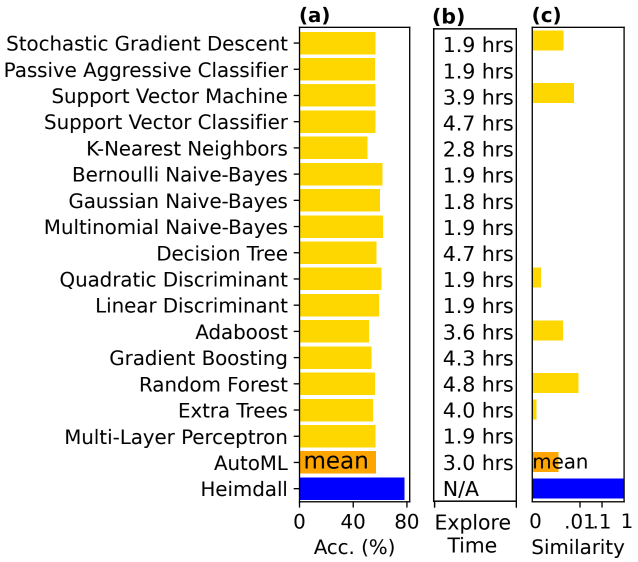

Figure 18. HEIMDALL vs. AutoML (§8). *Comparison of HEIMDALL and AutoML in terms of: (a) Accuracy, (b) Exploration time, (c) Model generalization.*

8.2 Towards Automatic HEIMDALL Pipeline

After understanding the effort in infusing domain knowledge to select and apply suitable techniques, one might wonder *if Automated Machine Learning ("AutoML") can replace our manual approach.* AutoML can indeed promote rapid design prototyping with less human labor. However, there are three limitations hindering AutoML's feasibility: (1) Limitation in understanding and applying domain knowledge, (2) Exploration computation overhead, and (3) Deployment considerations. To show these limitations, we use `auto-Sklearn` [25], a state-of-the-art framework that automates ML algorithm selection and its hyperparameter tuning.

In the first experiment, we randomly picked 50 datasets and supplied the *raw* dataset to the AutoML framework without the manual feature engineering steps that we did (§3.3). We use *16 AutoML classifiers* under various algorithm categories as shown in **Figure 18**. This is deliberate to see how much AutoML methods can help with the user exerting the least possible action.

Using `auto-Sklearn`, AutoML autonomously conducts hyperparameter tuning. **Figure 18a** compares the ROC-AUC (x-axis) of AutoML models (top rows) and HEIMDALL (last row). *AutoML models exhibit suboptimal performance, with an average accuracy 22% lower than HEIMDALL.* This outcome aligns with our expectation: given that AutoML exclusively utilizes the raw feature set without domain-specific derived features (*e.g.*, queue length), AutoML models are suboptimum in identifying meaningful correlations between the feature sets and the resulting label. Although one can configure the AutoML framework to explore derived features, the process will significantly increase the execution time.

In the second experiment, shown in **Figure 18b**, we highlight that *AutoML incurs high exploration time that leads to significant computational overhead during exploration.* Specifically, *the exploration time (in the x-axis) for AutoML models is ranging from 1.8 to 4.8 hours.* This is due to the inherently explorative nature and unbounded complexity of AutoML models. It is important to note that this experiment was conducted on a CPU, as `auto-Sklearn` does not support GPU acceleration by default. HEIMDALL's training time on the CPU could be further optimized (though beyond the scope of this paper) by porting the training code to C/C++ and leveraging CPU-optimized training libraries. Future work could explore optimizing AutoML models and reducing their training complexity, as their exhaustive exploration remains a challenge for efficient deployment.

In the third experiment (**Figure 18c**), we assess the generalizability of models generated by AutoML across different datasets. The results show that *AutoML creates models with highly divergent architectures for each dataset*, unlike HEIMDALL, which remains agnostic to the dataset. We quantify this lack of generalization by computing the cosine similarity of the models (log-scaled x-axis), where HEIMDALL consistently maintains a similarity score of 1. In contrast, *AutoML models demonstrate poor cross-dataset generalization (cosine similarities < 0.01).* This indicates that AutoML-generated models are not reusable across datasets, requiring re-exploration and retraining for each new workload, which incurs substantial costs to be deployed in production systems. All in all, although there is a significant potential for transforming HEIMDALL into a fully-automated system similar to AutoML, we leave this as future work.

9 Conclusion and Competitions

This paper presents HEIMDALL, an ML-powered I/O admission control designed with a strong emphasis on performance and deployment feasibility. HEIMDALL significantly reduces tail latency while maintaining sub-µs inference overhead. Our extensive evaluation demonstrates its effectiveness across multiple deployment environments, including user-level storage, in-kernel systems, and distributed storage clusters. Beyond its immediate impact on I/O admission control, HEIMDALL's extensible ML pipeline provides a foundation for broader research in ML-driven storage optimizations. To encourage further innovation, we have used HEIMDALL as a testbed for "mini competitions", inspired by popular competitions such as ImageNet and Kaggle [3, 4]. In one such effort, 15 students participated in exploring new techniques, experimenting with 20 forms of data augmentation, 35 classification models, and 20 regression models, as well as training strategies such as sampling and quantization. We release HEIMDALL code publicly, with the hope that it benefits the research community—not only in advancing I/O admission policies but also in exploring other critical storage optimizations and system-level ML applications.

Acknowledgments

We thank Yubin Xia, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. We also would like to thank Erci Xu for the helpful discussion and comments on this paper. This material was supported by funding from NSF (grant Nos. CCF-2119184, CNS-2402327, CNS-2027170, and CNS-2431425) as well as generous donations from Google, Meta, NetApp, and Seagate. The experiments in this paper were performed on Chameleon [2, 42]. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

A Artifact Appendix

A.1 Abstract

The artifact documents HEIMDALL's data science pipeline, *joint inference* technique, also client and kernel deployment.

A.2 Description & Requirements

There are 4 experiments outlined in experiments readme and all steps are given under *documentation* folder.

[How to access] The source code is publicly available in https://github.com/ucare-uchicago/Heimdall.git, DOI 10.5281/zenodo.14874299.

[Hardware dependencies] Client-level and kernel-level integration must be run on a machine with two unmounted SSDs, while others can be run on any machine.

[Benchmarks] We use public I/O traces from Microsoft [61], Alibaba [6], and Tencent [83]. These traces are a sample result of the preprocessing explained in Section §6.1.

A.3 Evaluation workflow

A.3.1 Major Claims. Note that due to the large scale experiment that took an ample amount of time, we show a random sample of small dataset for the artifact.

First, with several domain specific designs including period-based labeling and 3-stage noise-filtering, HEIMDALL's data science pipeline improves the decision accuracy from 63% to 93% compared to LinnOS (**Experiment (E1)**).

Second, with joint-inference, HEIMDALL maintains a low overhead with workload (4 mIOPS) that is 8 times heavier than the default one, while still maintaining accuracy greater than 80% (**Experiment (E2)**).

Third, HEIMDALL achieves lower average latency and tail latencies compared to other algorithms at the C client-level and using 500 workloads (**Experiment (E3)**).

Fourth, HEIMDALL achieves lower average and tail latencies at Linux kernel prototyping (**Experiment (E4)**).

A.3.2 Experiments. Each simulation experiment would take less than 30 human-minutes, and prototyping experiments can take up to 3.5 hours. We recommend running these experiments on Chameleon testbed.

*§**Experiment (E1):** Heimdall-Pipeline [15 mins human-minutes]: Evaluate the decision accuracy of HEIMDALL.*

[Preparation] Please follow the testbed reservation guideline to reserve a storage_hierarchy node at CHI@TACC site. Afterwards, follow the documentation.

[Execution] First, replay traces without the help of HEIMDALL and analyze the replayed results. We first compile the trace replayer and run sample traces. Then, we investigate into the traces we replayed. Tail analyzer script produces a profound characteristics profiling of replayed traces as part of our analysis. Next, we conduct period-based labeling, feature extraction and selection, and finally model training.

*§**Experiment (E2):** Joint-Inference [5 human-minutes]: Evaluate the accuracy-loss when utilizing joint-inference.*

[Preparation] The preparation phase of E2 is the same of E1 where the environment of E2 will be similar as E1.

[Execution] To prepare the training with joint-inference, in this step, we combine multiple I/Os into one I/O with extended features and an aligned label. We then train the model, deploy, and evaluate the accuracy on test set. You can change the value of `-batch_size` argument, observe the inference speed up and accuracy trade-off.

*§**Experiment (E3):** Client-Level [60 human-minutes]: Evaluate the resulting I/O latency and tail cut.*

[Preparation] Please reserve and set up a machine that have two unmounted SSDs from Chameleon Cloud.

[Execution] The documentation outlines step-by-step in running HEIMDALL against LinnOS, Random, Hedging, and LinnOS+Hedging. Training of HEIMDALL and LinnOS can be done in parallel while replaying has to be run *sequentially*. After running the script for each algorithm, run the script to analyze results and obtain the latency CDF graph.

*§**Experiment (E4):** Linux Kernel Deployment [60 human-minutes]: Evaluate the latency improvement*

[Preparation] Please follow the testbed reservation guideline. After cloning the artifact repository, follow the documentation to set up the machine.

[Execution] Compile the HEIMDALL kernel and update the grub configuration. Next, reboot the machine to let the compiled kernel be booted. After rebooting, you can check whether the kernel is successfully changed via *uname -r*, which is expected to output *6.0.0-heimdall*. To train the HEIMDALL module, we need to configure the devices and the traces to test with. Please change the configuration file in *$HEIMDALL_KERNEL/config/config.conf* on two variables:

- SSD_DEVICE0: unmounted SSD as primary replica.
- SSD_DEVICE1: unmounted SSD as failover replica.

After configuration, you can now train the model, replay the traces, and plot the results. You are expected to see the results of baseline (no I/O admission and prediction), random, and HEIMDALL. Two figures will be generated, one demonstrates the average read I/O latency and the other plots the latency percentiles.

References

[1] [n. d.]. Alibaba Block Traces. http://github.com/alibaba/block-traces.

[2] [n. d.]. Chameleon - A configurable experimental environment for large-scale cloud research. https://www.chameleoncloud.org.

[3] [n. d.]. Kaggle: Your Machine Learning and Data Science Community. https://www.kaggle.com/.

[4] [n. d.]. The Evolution of Image Classification Explained. https://stanford.edu/~shervine/blog/evolution-image-classification-explained.

[5] 2021. Colossus under the hood: a peek into Google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system.

[6] 2022. Alibaba Block Traces. https://github.com/alibaba/block-traces.

[7] 2024. Reflecting on 2023—Azure Storage. https://azure.microsoft.com/en-us/blog/reflecting-on-2023-azure-storage/.

[8] Najmeddine Abdennour, Tarek Ouni, and Nader Ben Amor. 2021. The importance of signal pre-processing for machine learning: The influence of Data scaling in a driver identity classification.. In *ACS 18th International Conference on Computer Systems and Applications (AICCSA)*.

[9] Ibrahim Umit Akgun, Ali Selman Aydin, Andrew Burford, Michael McNeill, Michael Arkhangelskiy, and Erez Zadok. 2023. Improving Storage Systems Using Machine Learning. In *ACM Transaction on Storage*.

[10] Jacob Alter, Ji Xue, Alma Dimnaku, and Evgenia Smirni. 2019. SSD failures in the field: symptoms, causes, and prediction models. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.

[11] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*.

[12] Shane Bergsma, Timothy Zeyl, Arik Senderovich, and J. Christopher Beck. 2021. Generating Complex, Realistic Cloud Workloads using Recurrent Neural Networks. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*.

[13] Andrew P. Bradley. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. In *The Journal of the Pattern Recognition Society Volume 30*.

[14] Kishan K. C., Rui Li, and Mahdi Gilany. 2021. Joint inference for neural network depth and dropout regularization.. In *Proceedings of the 35th Conference on Neural Information Processing Systems (NIPS)*.

[15] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. 2017. Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery. In *Computing Research Repository*.

[16] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Symposium on File and Storage Technologies (FAST)*.

[17] Zhen Cao, Geoff Kuenning, and Erez Zadok. 2020. Carver: Finding Important Parameters for Storage System Tuning. In *Proceedings of the 18th USENIX Symposium on File and Storage Technologies (FAST)*.

[18] Haihua Chen, Jiangping Chen, and Junhua Ding. 2021. Data Evaluation and Enhancement for Quality Improvement of Machine Learning. In *IEEE Transactions on Reliability*.

[19] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[20] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. 2020. Unearthing inter-job dependencies for better cluster scheduling. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*.

[21] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*.

[22] Anwesha Das, Frank Mueller, Paul Hargrove, Eric Roman, and Scott B. Baden. 2018. Doomsday: predicting which node will fail when on supercomputers. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.

[23] Jesse Davis and Mark Goadrich. 2006. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd International Conference on Machine Learning (ICML)*.

[24] Jeffrey Dean and Luiz Andre Barroso. 2013. The Tail at Scale. *Communications of the ACM (CACM)* 56, 2 (2013).

[25] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Proceedings of the 30th Conference on Neural Information Processing Systems (NIPS)*.

[26] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. 2023. Towards a Machine Learning-Assisted Kernel with LAKE. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[27] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[28] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[29] Klaus Greff, Antti Rasmus, Mathias Berglund, Tele Hotloo Hao, Jurgen Schmidhuber, and Harri Valpola. 2016. Tagger: Deep Unsupervised Perceptual Grouping. In *Proceedings of the 30th Conference on Neural Information Processing Systems (NIPS)*.

[30] Ajay Gulati, Arif Merchant, and Peter J. Varman. 2010. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*.

[31] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. 2017. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*.

[32] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. 2020. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*.

[33] Vikas Jaiman, Sonia Ben Mokhtar, Vivien Quema, Lydia Y. Chen, and Etienne Riviere. 2018. Héron: Taming Tail Latencies in Key-Value Stores Under Heterogeneous Workloads. In *The 43rd International Symposium on Reliable Distributed Systems (SRDS 2018)*.

[34] Virajith Jalaparti, Peter Bodík, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. 2013. Speeding up distributed request-response workflows. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[35] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. 2021. FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays. In *Proceedings of the 19th USENIX Symposium on File and Storage Technologies (FAST).*

[36] Wanchun Jiang, Yujia Qiu, Fa Ji, Yongjia Zhang, Xiangqian Zhou, and Jianxin Wang. 2023. AMS: Adaptive Multiget Scheduling Algorithm for Distributed Key-Value Stores. In *IEEE Transactions on Cloud Computing (TCC).*

[37] Wanchun Jiang, HaiMing Xie, Xiangqian Zhou, Liyuan Fang, and Jianxin Wang. 2019. Haste makes waste: The On-Off algorithm for replica selection in key-value stores. In *Journal of Parallel and Distributed Computing.*

[38] Wanchun Jiang, HaiMing Xie, Xiangqian Zhou, Liyuan Fang, and Jianxin Wang. 2019. Understanding and improvement of the selection of replica servers in key-value stores. In *Information Systems Journal (IS), Volume 83.*

[39] Michael I. Jordan and Robert A. Jacob. 1993. Hierarchical mixtures of experts and the EM algorithm. In *IEEE International Joint Conference on Neural Network (IJCNN).*

[40] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI).*

[41] Kapil Karkra. [n. d.]. Using Software to Reduce High Tail Latencies on SSDs. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180808_SOFT-201-1_Karkar.pdf.

[42] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, Françis Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC).*

[43] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab K. Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali Raza Butt. 2023. SHADE: Enable Fundamental Cacheability for Distributed Deep Learning Training. In *Proceedings of the 21st USENIX Symposium on File and Storage Technologies (FAST).*

[44] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. 2023. RECL: Responsive Resource-Efficient Continuous Learning for Video Analytics. In *Proceedings of the 20th Symposium on Networked Systems Design and Implementation (NSDI).*

[45] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. 2017. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST).*

[46] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An Informed Storage Cache for Deep Learning. In *Proceedings of the 18th USENIX Symposium on File and Storage Technologies (FAST).*

[47] Daniar H. Kurniawan, Levent Toksoz, Mingzhe Hao, Anirudh Badam, Tim Emami, Sandeep Madireddy, Robert B. Ross, Henry Hoffmann, and Haryadi S. Gunawi. 2021. IONET: Towards an Open Machine Learning Training Ground for I/O Performance Prediction. In *Technical Report University of Chicago TR-2021-03.*

[48] Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Ales Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. 2022. A continual learning survey: Defying forgetting in classification tasks. In *IEEE Transactions on Pattern Analysis and Machine Intelligence.*

[49] Timothée Lesort, Massimo Caccia, and Irina Rish. 2022. Understanding Continual Learning Settings with Data Distribution Drift Analysis. In *Computing Research Repository.*

[50] 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 Symposium on File and Storage Technologies (FAST).*

[51] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. 2021. IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern Flash Storage. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP).*

[52] Nanqinqin Li, Mingzhe Hao, Xing Lin, Huaicheng Li, Levent Toksoz, Tim Emami, and Haryadi S. Gunawi. 2022. Fantastic SSD Internals and How to Learn and Use Them. In *Proceedings of the 15th ACM International Systems and Storage Conference (SYSTOR).*

[53] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. 2016. PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proceedings of the 2016 EuroSys Conference (EuroSys).*

[54] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *Proceedings of the 21st USENIX Symposium on File and Storage Technologies (FAST).*

[55] Lei Liu, Xinglei Dou, and Yuetao Chen. 2023. Intelligent Resource Scheduling for Co-located Latency-critical Services: A Multi-Model Collaborative Learning Approach. In *Proceedings of the 21st USENIX Symposium on File and Storage Technologies (FAST).*

[56] Sidi Lu, Bing Luo, Tirthak Patel, Yongtao Yao, Devesh Tiwari, and Weisong Shi. 2020. Making Disk Failure Predictions SMARTer!. In *Proceedings of the 18th USENIX Symposium on File and Storage Technologies (FAST).*

[57] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-based Memory Allocation for C++ Server Workloads. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[58] Zheda Mai, Ruiwen Li, Jihwan Jeong, David Quispe, Hyunwoo Kim, and Scott Sanner. 2022. Online continual learning in image classification: An empirical survey. In *Neurocomputing.*

[59] Andrew McCallum. 2009. Joint inference for natural language processing.. In *Proceedings of the 13th Conference on Computational Natural Language Learning (CoNLL).*

[60] Rino Micheloni. 2017. Solid-State Drive (SSD): A Nonvolatile Storage System. In *2017 Proceedings of the IEEE (Proc. IEEE).*

[61] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. MSR Cambridge traces (SNIA IOTTA trace set 388). In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST).*

[62] Iratxe Nino-Adan, Eva Portillo, Itziar Landa-Torres, and Diana Manjarres. 2021. Normalization influence on ANN-based models performance: A new proposal for Features' contribution analysis. In *IEEE Access.*

[63] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P., Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J. R. Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *Proceedings of the 19th USENIX Symposium on File and Storage Technologies (FAST).*

[64] Jisung Park, Jeonggyun Kim, Yeseong Kim, Sungjin Lee, and Onur Mutlu. 2022. DeepSketch: A New Machine Learning-Based Reference Search Technique for Post-Deduplication Delta Compression. In *Proceedings of the 20th USENIX Symposium on File and Storage Technologies (FAST).*

[65] Mia Primorac, Katerina J. Argyraki, and Edouard Bugnion. 2021. When to Hedge in Interactive Services. In *Proceedings of the 18th Symposium on Networked Systems Design and Implementation (NSDI).*

[66] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*.

[67] Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. 2021. Scaling Vision with Sparse Mixture of Experts. In *Proceedings of the 35th Conference on Neural Information Processing Systems (NIPS)*.

[68] Liana V. Rodriguez, Farzana Beente Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning Cache Replacement with CACHEUS. In *Proceedings of the 19th USENIX Symposium on File and Storage Technologies (FAST)*.

[69] Sultan Mahmud Sajal, Luke Marshall, Beibin Li, Shandan Zhou, Abhisek Pan, Konstantina Mellou, Deepak Narayanan, Timothy Zhu, David Dion, Thomas Moscibroda, and Ishai Menache. 2023. Kerveros: Efficient and Scalable Cloud Admission Control. In *Proceedings of the 17th Symposium on Operating Systems Design and Implementation (OSDI)*.

[70] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. 2020. Activation Functions in Neural Networks. In *International Journal of Engineering Applied Sciences and Technology (IJEAST)*.

[71] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *Computing Research Repository*.

[72] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. A hierarchical neural model of data prefetching. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[73] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. 2023. LeaFTL: A Learning-Based Flash Translation Layer for Solid-State Drives. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[74] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*.

[75] Ruben van den Goorbergh, Maarten van Smeden, Dirk Timmerman, and Ben Van Calster. 2022. The harm of class imbalance corrections for risk prediction models: illustration and simulation using logistic regression.. In *Journal of the American Medical Informatics Association*.

[76] Shiva Verma. 2020. How to Create a Custom Loss Function | Keras.

[77] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*.

[78] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable and High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*.

[79] Gerhard Widmer and Miroslav Kubat. 1996. Learning in the presence of concept drift and hidden contexts. In *Machine Learning (ML)*.

[80] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S. Berger, Nathan Beckmann, Gregory R. Ganger for ML admission, and cache prefetching. 2024. Baleen: ML Admission & Prefetching for Flash Caches. In *Proceedings of the 22nd USENIX Symposium on File and Storage Technologies (FAST)*.

[81] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*.

[82] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. 2023. GL-Cache: Group-level learning for efficient and high-performance caching. In *Proceedings of the 21st USENIX Symposium on File and Storage Technologies (FAST)*.

[83] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. 2020. Tencent block storage traces (SNIA IOTTA trace set 27917). In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*.

[84] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. 2022. Aequitas: admission control for performance-critical RPCs in datacenters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[85] Alice Zheng and Amanda Casari. 2018. Feature engineering for machine learning: principles and techniques for data scientists.

[86] Xianqian Zhou, Liyuan Fang, HaiMing Xie, and Wanchun Jiang. 2019. TAP: Timeliness-aware predication-based replica selection algorithm for key-value stores. In *Concurrency and Computation: Practice and Experience (CCPE'19), Volume 31*.

[87] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, More Harchol-Balter, and Gregory R. Ganger. 2014. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*.

[88] Lorenzo Zuolo, Cristian Zambelli, Rino Micheloni, Davide Bertozzi, and P. Olivo. 2014. Analysis of reliability/performance trade-off in Solid State Drives. In *IEEE International Symposium on Reliability Physics (IRPS)*.