# Using Machine Learning to Predict Optimal Erasure Coding Policies for Object Storage System in OpenStack Swift

by

Amio Malakar Ankon
21201669
Boloy Chaki
21301205
Mashrur Shakhawat Sayan
21201221
Debashish Sarker
21201580

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science and Engineering

Department of Computer Science and Engineering
Brac University
February 2026.

# Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

**Student's Full Name & Signature:**

| | |
|---|---|
| Debashish Sarker<br>21201580 | Amio Malakar Ankon<br>21201669 |
| Boloy Chaki<br>21301205 | Mashrur Shakhawat Sayan<br>21201221 |

# Approval

The thesis titled "Using Machine Learning to Predict Optimal Erasure Coding Policies for Object Storage System in OpenStack Swift" submitted by

1. Amio Malakar Ankon (21201669)
2. Boloy Chaki (21301205)
3. Mashrur Shakhawat Sayan (21201221)
4. Debashish Sarker (21201580)

of Fall, 2024 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science in February, 2026.

**Examining Committee:**

Supervisor:
(Member)

Dr. Jannatun Noor Mukta

Associate Professor, CSE
Director, BSDS
United International University

Thesis Coordinator:
(Member)

Md. Golam Rabiul Alam, PhD

Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Sadia Hamid Kazi

Associate Professor and Chairperson
Department of Computer Science and Engineering
Brac University

# Abstract

Erasure coding helps to reduce storage overhead and improve fault tolerance. But the procedure to select an appropriate erasure coding policy is complex, which often involves tradeoffs among various metrics such as access latency, recovery behavior, storage efficiency, etc. Generally, these are handled using static or heuristic-based configurations, which can not account for variations in workload. In the industry, service providers like Ceph do benchmarking based on throughput/latency without considering workload diversity. OpenStack Swift, one of the most widely used open-source object storage systems, supports erasure coding, but it allocates policy selection in a manual, static way - without it being workload-aware. To address this problem, this thesis showcases a data-driven performance modeling framework for erasure coding in object storage systems using machine learning. A structured dataset- ABDS-30k was constructed in a controlled execution of varying workload conditions. It was created on a Swift All-In-One (SAIO) testbed under different erasure coding policies with the addition of failure injections. The dataset collects several empirically observed performance metrics such as read and write latency, tail latency, success rate, and reconstruction time in case of disk failures. For the job of selecting the optimal erasure coding policy, we adopt two distinct Machine Learning paradigms - a regression-based approach of performance modeling and a classification-based approach for direct data-driven policy recommendation. For regression, CatBoost, XGBoost, and Random Forest Regression are used to predict target metrics in a given workload context and failure scenario, and the optimal policy is chosen based on a weighted score. In the classification-based approach, CatBoostClassifier, XGBoostClassifier, and Logistic Regression are used to label workload–policy pairs using an oracle cost function, and the most optimal erasure coding policy is directly recommended. Experimental results show that the regression models have moderate prediction errors for latency and recovery metrics because of the inherently noisy and heavy-tailed nature of the system, but they remain effective in optimal policy recommendation by exhibiting top-1 accuracy with 48.16% in XGBoost Regression and a top-3 accuracy 100% in Random Forest Regression model. Also, the regret mean (0.010 - 1.81) and regret median (0.00216 - 0.089) values showed a very low margin of error. Classification-based approach shows comparatively weaker metrics, indicating that it is not optimal for data-driven policy recommendation. Overall, this shows the feasibility of using ML-based performance modeling as opposed to static and heuristic-based policy selection, which can be later used as a foundation for future control-plane automations.

**Keywords:**
Cloud Computing, Object Storage System, Machine Learning, Erasure Coding, OpenStack Swift

# Acknowledgement

We sincerely thank all who have supported us throughout this journey. Firstly, we appreciate our supervisor, Jannatun Noor, for her direction, support, and encouragement throughout the entire process. Her guidance and valuable feedback were significant in shaping this research, and we sincerely appreciate her time and dedication. Finally, we would like to extend our gratitude to Md. Abu Obaida Zishan and Md. Farhadul Islam for their support and guidance throughout the process.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Background

In the current data-driven world, most business models rely on cloud storage, increasing the need for cost optimization. Reduction of latency, the delay in data transfer, is essential for enhanced user experience. In cloud architecture, to meet these challenges, object storage systems are becoming famous day by day, which is a type of data storage architecture used to manage and store unstructured data while ensuring scalability. This considers units of data as objects that contain raw unstructured data in the form of images, videos, etc, and metadata, which is information about the object and a unique global identifier. In today's day and age, all intelligent devices and machines create insane amounts of data, such as cookies, data packets, information, etc. These data are constantly driving the servers towards the edge of their capacity, often overwhelming and sometimes even hanging up whole complete systems. These create bottlenecks in the system that normally slow down the complete system and in severe cases, may even cause the loss of important data connections and packets. According to Lui et al. [26], these bottlenecks bring forth the necessity of reducing storage overhead and efficiency- two key aspects essential to improving responsiveness and smooth data flow in a system. OpenStack Swift, an object storage system, uses erasure coding, a novel method to store data, which is an alternative to replication and has lower storage overhead and fault tolerance. In OpenStack Swift, erasure coding policies are administratively assigned and implemented when objects are uploaded to the system. But, the selection policy of erasure coding policies are not simple and does not strictly depend on partitions, with various metrics like read and write latency, tail latency, success rates, reconstruction time with many more factors which depend on workload and system configuration. In this paper, we create a system based on machine learning that models object upload, download, and reconstruction behavior under varying workloads and failure conditions, and recommends an optimal erasure coding policy using a weighted, performance-aware scoring strategy.

In current IoT and cloud computing, erasure coding has become an indispensable part of distributed storage systems because of its effectiveness in data distribution and durability with robust data [39]. Erasure coding stores data by encoding it and partitioning it into k+m cells, where k is the number of data cells, and m represents the number of parity cells [19]. Although this achieves better space efficiency than

replication, performance degradation still happens when the volumes of k and m increase [19]. So, erasure coding by itself is not always perfect, and the selection of erasure coding can not always be determined by the combination of k & m. In our work, we address this gap by evaluating EC policies in an OpenStack Swift testbed by executing numerous workload scenarios and building a machine learning model that helps to determine the optimal EC policy.

Most of the existing research and literature have evaluated EC schemes only within a few controlled workload variations and work on improving EC itself based on its fault tolerance, robustness tolerance etc metrics. Some papers, such as MSR Cambridge, Harvard NFS provide traces for workload realism for EC evaluation (Chan et al., 2014) [5], but they do not provide an explicit overview of erasure coding policy per workload. So we created our own dataset by using OpenStack Swift and testing EC coding policies in various workloads, and creating an ML-based system that predicts optimal policy. EC in OpenStack Swift uses actual fragmentation, encoding, and decoding, so our dataset contains real client I/O and backend work metrics.

In the context of erasure-coded storage systems, relationships between configuration parameters, workloads, failure conditions, and performance metrics are highly complex. Traditional static policy selections cannot fully ensure the optimal policy selection due to the intricate nature of the dependencies. By using ML techniques on performance data, we can create a predictive policy recommendation system from system state and workload features. This can allow administrators to find the optimal policies for their specific efficiency, latency, and robustness needs.

## 1.2 Rational of the Study or Motivation

In the current world of IoT, it is estimated that 2 billion terabytes of data will be stored in cloud storage by 2025, and by 2029, the cloud market is projected to be worth 376.36 billion USD [38]. This showcases the significance of storage policies in object storage systems to keep up with the market. Additionally, Gartner forecasts that worldwide public cloud end-user spending will reach $723.4 billion in 2025 [29]. This shows the rapid growth of cloud adoption and the increasing economic importance of efficient storage management. In systems like OpenStack Swift, with each new workload, the configuration has either to be tested empirically for policy selection or chosen based purely on partition numbers- which is not always optimal. The main motivation behind this project is to create a framework that suggests an optimal EC policy given a specific workload scenario based on authentic data. Using a machine learning framework can give proper assessment of optimal policy of a system within milliseconds where real system simulation and checking often cost time and money.

## 1.3 Problem Statement

In the industry, service providers like Ceph [4] do benchmarking based on throughput/latency without considering workload diversity. OpenStack Swift [41] itself

allocates erasure coding policies administratively by creating a ring and disks in a static, manual method without the scope considering any workload context beforehand.

In addition to the EC configuration, an object storage system has numerous factors that determine the upload, download latencies, and reconstruction time of objects stored in it. While various models for enhancing storage performances exist, there is a lack of authentic data-driven models that can evaluate all relevant metrics related to EC system and recommend optimal policies based on the specific workload context.

So, this makes it impossible to systematically recommend an optimal policy without empirical testing for each new workload. Therefore, we address this core problem by suggesting a machine learning based framework to optimally select erasure coding policies for varying workloads and mitigating the limitations in static configurations.

## 1.4 Objective

The primary objective of this study is to develop and evaluate a machine learning based framework that can suggest optimal erasure coding policies in OpenStack Swift. To achieve this objective:

- A controlled OpenStack Swift All-In-One (SAIO) cluster is created and a workload generator is developed to generate benchmark workloads across the configured erasure coding policies.
- The workload generator is executed across the configured EC policies and under induced failure conditions to create a structured dataset suitable for machine learning.
- Using the created dataset, separate classification and regression models are trained and validated for suggesting optimal EC policy.

Thus, we evaluate the overall feasibility of the ML-based approach of finding an optimal EC policy for a workload context.

## 1.5 Methodology in Brief

In this research we aim to propose a machine learning framework that can suggest the optimal erasure coding policy based on workload context in object storage systems. For this we simulate authentic workload contexts in a testbed and create a dataset, then train and evaluate the machine learning models on suggesting the optimal EC policy.

1. OpenStack Swift is used as a testbed to simulate and generate the workload contexts, and the dataset - ABDS-30k is generated based on that.

2. The dataset is used as an input for the machine learning framework, where two separate paradigms are used based on classification and regression.

3. Both the regression and classification-based models are trained on the dataset and predict the optimal policy for a specific workload context based on a specific score, which indicates the upload, download latency, and object reconstruction time.

4. The best optimal policy is recommended to the user, which may further change in the future based on the then-current usage of the user and the workload context.

## 1.6 Scopes and Challenges

This study aims to design and evaluate a machine learning-based framework for predicting optimal EC policies in object storage systems. As cloud storage platforms continue to grow in scale and to make sure that storage space is used perfectly, it is important to select the correct EC policy for balancing storage efficiency, access latency, and fault tolerance. While erasure coding offers superior storage efficiency, saving storage space over the replication method, different EC configurations have different performance and recovery trade-offs, making a complex policy selection for a dynamic environment. Below, we discuss the scopes and challenges regarding our research.

**Scopes:**

1. **Automated EC Policy Selection:** For predicting the most suitable erasure coding policy, a machine learning model is trained on a dataset based on system metrics for different erasure coding policies. This reduces the reliance on manually tuned configuration to check manually which policy is optimal.

2. **Scalability in Large-Scale Object Storage:** For cloud data centers or platforms, the proposed approach can scale with an increased number of storage nodes, objects, and their sizes.

3. **Improved Resource Utilization:** The proposed framework supports a resource-aware EC policy prediction system by understanding workload context behavior. This indirectly improves storage efficiency and reduces unnecessary I/O overhead without modifying OpenStack Swift's internal mechanism.

4. **Adaptation to Workload:** An ML-based policy prediction or selection mechanism can adapt to changing access patterns, object sizes, and failure conditions.

5. **Cost-Aware Storage Management:** With optimal EC polices, cloud data centers can save on storage and improve recovery costs with better performance.

6. **Decision Support for Administrators:** Storage administrators in cloud platforms can understand EC trade-offs. So, the proposed model acts as a decision support tool.

**Challenges:**

1. **Dataset Creation:** Creating a dataset that covers multiple EC policies and different workload scenarios with system metrics is very challenging.
2. **Performance Overhead:** Collecting system metrics and ML inference introduces overhead.
3. **Trade-off Modeling Complexity:** EC policies involve complex trade-offs between latency, bandwidth, and storage overhead. Cloud storage cost–performance optimization is a major challenge, so finding globally optimal configurations can be difficult [28].
4. **Machine Learning Model Selection:** To select the optimal EC policy, two machine learning paradigms- classification and regression- are explored. Since these paradigms have different workflows and results, selecting and comparing them introduced additional modeling and interpretation challenges.
5. **Security and Privacy Risks:** Compression can reduce the size of data packets, but poorly planned methods can expose patterns making the whole model vulnerable to inversion attacks [6].
6. **Device Heterogeneity:** The hardware and network characteristics of the environment used for dataset generation can significantly influence the observed performance metrics. CPU capacity, memory, and network bandwidth may bias the collected data, potentially limiting the generalizability of machine learning models that will be trained.
7. **Algorithm Complexity:** Implementation of complex algorithms can slow down the whole model or take up too many resources.

## 1.7 Key Learnings and Insights

In this study, we get valuable insight into the object storage system and trade-offs involved in erasure coding policy selection. Our experimental results demonstrate that no single EC policy is suitable or optimal across all workload and system conditions. The performance and cost-outcomes depend on the workload characteristics. ML can be implemented to ensure that it effectively models these complex relationships to improve storage efficiency and select an optimal EC policy.

The performance and cost outcomes are dependent on object size, access frequency, network bandwidth, and failure states. A key insight of this work is that ML can predict suitable EC policies based on observed system and model these complex relationships. This does not replace existing storage mechanisms, but it can enhance

decision-making by data-driven policy selection.

The findings highlight that ML-based EC policy prediction can improve storage utilization and reduce operational cost while making sure the system is performing better. In the industry, in platforms such as Ceph and OpenStack Swift, EC policy decisions are typically guided by empirical performance evaluation, workload analysis, and heuristic trade-offs rather than dynamic optimization [40]. From our research, we find that ML-based performance modeling is feasible for predicting optimal EC policies for a workload context, which can be later used as a foundation for future control-plane automations.

## 1.8 Thesis Organization

This thesis report has been organized in a sequence that gradually moves from motivation of the research to implementation, evaluation and future work. The chapters of our thesis paper is listed below:

- Chapter-1 gives the core overview of the whole thesis workflow with an introduction to concepts of object storage system and erasure coding and explains the core research gap of EC policy selection- and how it is addressed in this research.
- Chapter-2 showcases a complete literature review on the core works done in this sector, which includes object storage performance, erasure coding trade-offs, benchmarking various schemes and different optimization techniques.
- Chapter-3 describes the system requirements and design specifications with explicit descriptions of the assumptions as well as the practical constraints of this framework environment.
- Chapter-4 shows the full dataset creation procedure with all the necessary steps of environment creation in the testbed with detailed explanation of the workload variations and failure injection strategy.
- Chapter-5 covers the full methodology from data preprocessing to training and performance modeling of both machine learning paradigms.
- Chapter-6 showcases the experimental results and full analysis of the policy recommendation pipeline.
- Chapter-7 concludes the study with the key findings, future steps, contribution and limitations.

# Chapter 2

# Literature Review

## 2.1 Preliminaries

This literature review primarily focuses on research related to object storage systems, erasure coding mechanisms, and the application of machine learning techniques for optimizing storage policies in cloud environments.

### 2.1.1 Object Storage System

Many cloud object storage systems have become fundamental infrastructure for modern analytical database systems [25]. An object storage system organizes data into immutable objects stored within a container called buckets. Each object has its own unique identifier and metadata. Object storage systems use a distributed architecture with no central point of control, providing greater scalability, redundancy, and permanence. Object storage is ideal for cost-effective and scale-out storage. There are many object storage systems like AWS S3, Google Cloud Storage, OpenStack Swift, etc. OpenStack Swift provides a fully RESTful API, enabling programmatic access to scalable and redundant object storage.

### 2.1.2 Erasure Coding

Erasure coding (EC) is a method of data protection used in a distributed storage system to minimize storage overhead [3]. Unlike the replication method, which stores a full copy of data in multiple storage locations, erasure coding divides data into fragments, adds redundant parity fragments, and distributes them across multiple nodes. Even if a drive fails or data becomes corrupted, the data can be reconstructed from segments that are stored on other drives. Here, the original data is split into k data fragments and generates m parity fragments. So, the total number of fragments is n=k+m. Erasure coding saves storage space significantly and provides higher fault tolerance by allowing data recovery from multiple concurrent failures. Even with many advantages, this has a computational overhead for encoding and decoding with reconstruction latency.

### 2.1.3 XGBoost

XGBoost, or Extreme Gradient Boosting, is a scalable end-to-end tree boosting system that is used in machine learning models. Here, each tree is trained to correct the errors of the existing ensemble [7]. To learn the set of functions used in the model, it minimizes a regularized objective function that combines a loss function with complexity penalties.

**Tree ensemble formulation:** Given an input feature vector $x_i \in R^m$, XGBoost predicts an output by summing the contributions from $K$ trees:

$$\hat{y}_i = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i), \qquad f_k \in \mathcal{F} \tag{2.1}$$

where each $f_k$ is a regression tree that maps $x_i$ to a leaf index and outputs a leaf weight. In regression, the target $y_i$ is continuous (e.g., latency in milliseconds). XGBoost typically uses losses such as:

- **Squared error:** $\ell(\hat{y}, y) = (\hat{y} - y)^2$.

The model output $\hat{y} \in R$ directly. Because the tree learners are regression trees and the ensemble sums real-valued leaf weights.
**Regularized learning objective:** XGBoost trains trees by minimizing a regularized objective:

$$\mathcal{L}(\phi) = \sum_i \ell(\hat{y}_i, y_i) + \sum_k \Omega(f_k), \qquad \Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2. \tag{2.2}$$

In binary classification, the label $y_i \in \{0, 1\}$. XGBoost commonly uses logistic loss, where the model produces a score and converts it to a probability:

$$p_i = \sigma(\hat{y}_i) = \frac{1}{1 + e^{-\hat{y}_i}}. \tag{2.3}$$

Training then minimizes a classification loss using the same regularized boosting framework.

### 2.1.4 CatBoost

CatBoost is a powerful machine-learning technique that can handle categorical features natively without one-hot encoding [14]. It overcomes common prediction bias issues in traditional boosting methods by introducing two algorithmic contribution: ordered boosting and ordered target statistics. For the regression problem in CatBoost, the target is continuous, $y \in R$, so the model is trained to minimize a regression loss—most commonly Mean Squared Error (MSE)—defined as $L(y, \hat{y}) = (y - \hat{y})^2$.
For the classification setting in CatBoost, the target is binary, $y \in \{0, 1\}$, so the model is trained to estimate class probabilities by minimizing the log-loss objective:

$$L(y, \hat{p}) = -\Big[y \log(\hat{p}) + (1 - y) \log\big(1 - \hat{p}\big)\Big], \tag{2.4}$$

where $\hat{p}$ predicts the probability of the positive class.

### 2.1.5 Logistic Regression

Logistic Regression is a statistical classification method used to model the probability [33]. It is a supervised learning algorithm used for classification tasks. The probability is a binary outcome based on the predictor variables. The dependent variable Y represents the outcome as 0 or 1. The probability model for logistic regression $\pi_i$ is formulated as

$$\pi_i = \frac{\exp(\beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n)}{1 + \exp(\beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n)} \tag{2.5}$$

To adjust into linear function, a logit transformation is applied:

$$logit(\pi_i) = \log\left(\frac{\pi_i}{1 - \pi_i}\right) = \beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n \tag{2.6}$$

**where:**

- $X_j$ denotes the predictor variable (feature),
- $\beta_j$ represents the regression coefficient,
- $n$ is the number of features.

### 2.1.6 Random Forest Regression

Random Forest is a machine learning algorithm that is used for both classification and regression. According to Salman et al. [34], Random Forest regression operates by splitting into many trees during training and gather average prediction from the individual trees as the final regression value. Each tree is built by bootstrap aggregation and random forest selection. Bootstrap creates K samples from the dataset. For each sample, the regression tree splits nodes based on features that minimize the variance of the target variable. Final output is the mean of all the tree prediction where for a new input, each tree gives a prediction.

### 2.1.7 Evaluation Metrics

**Mean Absolute Error (MAE):** This is the average magnitude of error between actual and predicted values in dataset [15].
Below is the mathematical formula of this metric:

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i| \tag{2.7}$$

**Where:**

$$\hat{y} = \text{predicted value of } y$$

$$\bar{y} = \text{mean value of } y$$

**Root Mean Squared Error (RMSE):** To understand RMSE we need to understand MSE [15]. This is the average of the squared difference between the original and predicted values in the dataset. RMSE is the root of MSE. It measures the

standard deviation of residuals.
Below is the mathematical formula of this metric:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{2.8}$$

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2} \tag{2.9}$$

**The coefficient of determination:** $R^2$ is the coefficient of determination. This indicates how much of the variance in the dependent variable is explained by the linear regression model. $R^2$ is a scale-free score and typically ranges from 0 to 1 in practice [15].
Below is the mathematical formula of this metric:

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2} \tag{2.10}$$

## 2.2 Review of Existing Research

The following literature review contains a detailed overview of the various methodologies tried and tested to optimize object storage systems, erasure coding, and machine learning.

Noor et al. [39] benchmark multiple erasure coding schemes to evaluate time efficiency and fault tolerance in the OpenStack Swift object storage system, specifically on the three Solomon-Reed schemes. The methodology revolves around 2 testbeds, both local and remote, along with the SimEDC simulator to establish a benchmark (COCO-17) for a custom dataset (MCSD-100). One of the key findings in this study is that if the number of fragments and parity data is increased, it concurrently leads to longer upload, download, and deletion time being the downside. Thus, highlighting a core performance bottleneck. Another research gap exposed by this study is the performance of EC in cloud storage. However, this study primarily investigates the efficiency tradeoff between I/O performance without extensive testing on real-world dynamic cloud situations. The author concludes the study with the suggestion of future research on load balancing and on adaptive coding strategies to optimise the tradeoff between efficiency and performance.

A study by Pei et al. [8] proposes Group U, which is a grouped and pipeline scheme that is designed to update in erasure-coded distributed systems. To counter high I/O overhead and low efficiency, this research proposes a radical new idea of using a hybrid three-layer framework that would support both single and multiple updates. One of its main innovations is to include a workload-aware grouping system based on node groups and pipelined techniques for parallelized data transmission and communication. The evaluation on RDFS shows time reduction by 21%-69% and overhead reduction by 22%-46% compared to PUM and PDP-P. The study was

done in a controlled environment, which keeps the door open for a real-world validation.

A novel approach is proposed by Hu et al. [31], where they developed high-performance erasure coding libraries by leveraging existing optimized machine learning (ML) libraries rather than building custom implementations. According to the authors, erasure coding is a key technique for fault-tolerant storage. They claim that EC can be easily implemented via ML libraries; EC implemented via ML libraries can be executed on a variety of hardware platforms. To support their claims, they developed EC libraries using Apache TVM. Traditional EC libraries adapt to heterogeneous platforms, including GPUs, CPUs, and accelerators, and demand a high level of expertise in both coding theory and low-level hardware optimization. ML libraries allow developers to reuse decades of optimization effort and reduce development complexity. They mapped erasure coding to general matrix multiplication (GEMM), showing that bitmatrix coding is structurally identical to GEMM. Relying on TVM's auto-tuning and hardware optimization, they defined the erasure coding within 40 lines of codes and they compared TVM-EC against state-of-the-art libraries on a CPU platform using the Reed-Solomon algorithm. With that, they achieved up to 1.75x higher throughput than custom libraries. All erasure coding libraries achieved lower encoding throughput with higher values of the parameters. There is also a limitation in their work where they pointed out that ML libraries may perform badly with hardware with few parallel computing resources. Also, they face integration complexity because ML libraries expose a high-level API, making integration with low-level storage system non-trivial. The authors demonstrated that reusing optimized ML libraries for erasure coding is a promising approach, and this significantly reduces development burden while achieving superior performance. For their future work they want to investigate energy and latency trade-offs.

Mondal et al. [20] propose a ML based classification for storage space recommendation in an object storage system in cloud computing. They focus on RSoS, an object-based schema-oriented data storage system, which handles various types of data across multiple database backends. Their motivation is to evaluate a classification framework that predicts the appropriate storage location for incoming data. In healthcare, incoming data is unstructured and comes from various sources, and manually configuring storage metadata is error-prone. So the authors identify the gap and want the system to automatically classify and route multi-format data to schema-appropriate storage using behavioral metrics in real-time. They use Ganglia to collect system metrics each time a query hits RSoS. For the feature selection, they compared three supervised algorithms and tested three different classifiers. They achieved 100% accuracy, precision, and recall in predicting storage location for three data types using LII21 + K-NN. The system correctly separated data types in the databases. Looking at down side, they only validated on three different healthcare data types, so other types of data, like video and image is not used, resulting in a lack of proof in these sorts of file types. But their proposed method reduces administrative overhead and improves latency over manual methods. Their next aim is to extend the functionality of the classification engine in such a way that it can provide more prominent results in less time.

A study by Syed and Albalawi [35] tries to use machine learning to efficiently allocate resources for cloud computing services. A Kaggle dataset is used here for predictive analysis. Cost adaptation for real-world applications was also key here. They are using prior history and cloud scalability to train the model to give them a machine-learning model that would optimize cloud resource management, using predictive models, optimization algorithms, and real-time adaptation. The study has some failings as it does not have any real-world deployment. It also has unreliable data quality with challenges in scalability with inconsistent edge case handling with overreliance on the Kaggle dataset.

A paper by Chen et al. [30] tries to use a hybrid storage system for cloud-scale object storage to maintain the metadata. This hybrid model uses a mix of SSD and HDDs to access CEPH and OASIS models. Here they are using SSD for bigger data and frequently accessed data and HDD for smaller ones. This storage is also further compressed in MDS and OSD clusters. They use the RADOS service through the CLUSTER algorithm to complete this. Here cold metadata is transformed into hot metadata by metadata Migration. This model outputs lower performance efficiency when combining hot metadata and Data is disabled.

A research paper by Noel et al. [13] developed an Adaptive Resource Management (ARM), which uses a Reinforcement Learning (RL)-based approach. It enables the self-managing cloud storage system by automatically balancing load and migrating data in response to performance hotspots. They implemented and evaluated on Ceph, an open-source distributed object storage system. They collected system metrics, used a stochastic policy gradient RL algorithm to decide which one to take action in the correct time, and executed the action by adjusting Ceph's OSD weights. The performance monitor fetches the system data from the Ceph OSDs at 30 seconds. They achieved up to 50% faster read response and 33% faster write response time compared with the default Ceph. They also outperformed a state-of-the-art dynamic load rebalancing technique by 43% read response time and 33% write response time. Also, in their network interference, the performance of ARM is similar to Ceph's default case, except for one workload, where ARM takes the necessary actions to improve the performance of read response time up to 42% compared with Ceph's default and 44% compared with DLR, respectively. This paper demonstrates that RL can overcome the limitations of manual tuning and it can automate cloud storage performance management.

A study by Akbar et al. [24] addresses the challenges of data deduplication in a cloud storage system. In this paper, the authors want to increase the reliance on cloud service demands efficient storage and data management, where big data contains redundant information from multiple sources. So removing these duplications can save storage space and bandwidth. That's why they proposed a hybrid ML-based approach to achieve efficient deduplication and increase security by securing data through encryption, which conflicts with deduplication in cloud environments. They use Two Threshold Two Divisor (TTTD) with Dynamic Prime Coding (DPC) to chunk data based on content. This improves the deduplication ratio and reduces processing time. They use a multi-level hashing technique to employ multiple hash functions to reduce CPU overhead. They also implement the Chameleon Hash

Signature Algorithm and Blacklist Merkle Tree for security. They achieved an improved deduplication ratio by higher redundancy elimination compared to fixed-size and CDC methods. They enhanced throughput, where they achieved 200 KB/s in chunking throughput. Hash function computation was reduced to 772 ms, making it lower in computational cost, and optimizing the chunk size saves significant storage in post-deduplication. Their proposed hybrid deduplication system outperforms existing methods in terms of ratio, throughput, and computational cost while maintaining security. For future work, they want to work on ML for anomaly detection and integrate advanced encryption to protect data.

A paper from Durner et al. [25] gives us a brief description of how cloud object storage can be used efficiently and cost-effectively by presenting a cloud object storage download manager with a brief analysis of cost, latency, throughput, and performance compared to state-of-the-art database systems. Cloud object storage faces challenges like high latency due to many concurrent requests. High CPU overhead is caused by a large set of data that needs to be analyzed simultaneously and Multicloud supports complexities, as different cloud vendors provide their own network libraries, making it difficult for users to manage multiple cloud databases efficiently. They are proposing an approach with AnyBlob and integration with Umbra DBMS. AnyBlob reduces CPU overhead and reduces latency. It uses an asynchronous I/O system to manage multiple connections in a thread. It has 3 major components which are io_uring low-overhead system call interface, state machine-based message, and asynchronous system call. From performance evaluation, experiments were conducted on AWS with different settings. They achieved maximum performance using 0.7 times the CPU resources and 1.5 times the performance under a fixed CPU budget with an additional 10% reduced CPU usage showing that AnyBlob improves the efficiency of object cloud storage by optimizing resource utilization and reducing computational overhead.

A study by Bucur et al. [11] provides a review of object storage in single-cloud and multi-cloud environments. This highlights the lack of unified storage solutions across major providers such as Amazon S3, Azure Blob Storage, and Google Cloud Storage. In this paper, the analysed individual cloud providers compare their performance, architectural approaches, and storage classes. They also overviewed multi-cloud storage, their challenges, like security, multiple SLAs, and implementation. To address these issues, they propose the development of autonomic SLA management systems, vendor-agnostic APIs, and secure data distribution models. So, their review focuses on multi-cloud storage that can enhance security, availability, and vendor diversification. However, their study does not provide cost optimization and integration strategies for practical deployment.

A study by Wang et al. [36] addresses the demands for computational and storage resources, because cloud computing and machine learning algorithms are evolving. The global cloud computing market is expanding rapidly and ML applications are taking over industries, creating unusual resource requirements. The authors proposed using cloud platforms to optimize ML workloads, only focusing on storage efficiency in distributed file systems. They argue that traditional storage systems struggle with memory limitations and garbage collection. They say that ML can

optimize metadata handling, improve data retrieval, and boost system performance. Using DanceNN, a universal distributed directory tree service and Random Forest model for performance prediction, can optimize a hybrid cloud-ML framework. This framework is centered on metadata optimization. From their experiment results, DanceNN eliminates memory bottlenecks, and the Random Forest model identifies performance patterns and potential bottlenecks. From their research, the integration of ML and cloud computing can enhance metadata scalability and storage optimization. The DanceNN and Random Forest approach shows the improvement of system performance.

A study by Matt et al. [9] addresses the challenges of efficiently storing data across multiple cloud storage providers. They presented a heuristic optimization approach that optimizes storage placement, cost, and latency efficiently while considering user-defined quality of service (QoS) requirements. Their system, implemented within the middleware CORA, uses erasure coding to split objects into chunks. These chunks are distributed on different cloud services to evaluate them based on pricing models, availability, access pattern, and latency data. In their evaluation setup, they used access logs from UbuntuOne over 1M+ users within 30 days and observed many metrics such as cumulative cost, upload/download latency, optimization duration, and metadata overhead. This demonstrates cost saving up to 30% and latency reduction of up to 48% in upload time and 69% in download times. Their proposed approach reduces cost and latency for redundant cloud storage while meeting QoS requirements.

A study by Factor et al. [1] provides the concept of an object storage system. They recommend object storage for its role as the future building for large-scale storage systems, research prototypes, and industry options. They point out the T10 OSD protocol (version 1) as a self-managed storage access via cryptographically enforced capabilities. They showed that T10-compliment object storage controller achieves throughput up to 109 MB/s cache-hits read operations on a 1 Gbit Ethernet switch. While object storage offers advantages in decentralized space management, metadata integration, and security, its adoption is slowed down by a paradigm shift from block-based interfaces. This paper concludes that open-source tools are essential for accelerating the adoption of object storage systems.

A study by Su et al. [10] proposes an analytical performance model to predict response latency for cloud object storage systems. They argue that response latency is a vital component for cloud providers and their users, which also directly impacts revenue. They propose a queuing-theory-based performance model to predict response latency percentiles for systems, focusing on event-driven-architecture and complex disk operations. For their modeling approaches authors packed multiple disk operations into a single "union operation" to fit the queuing model. For their backend processes, they use M/M/1/K queue approximation for disk I/O queuing, and for total latency modeling, they use three-tier latency composition. For parameter estimation, they uses Gamma distribution to fit the disk service time to measure and collect system metrics from monitoring tools. From their experiments, they achieved 4.44% prediction error in their model, but in their variety scenarios, they reduced the prediction errors by up to 73% compared to baseline models. From

their research, they validated on OpenStack Swift, enabling efficient capacity planning and SLA compliance.

A study by Kamble [32] focuses on resource allocation using CNN, LSTM, and Transformer deep learning algorithms that help the Cloud Service Providers (CSP) to improve cost-effectiveness, performance, and resource utilization. The authors address practical issues like cost, performance, and operational efficiency while leveraging real-world datasets (Google Cluster Data) for credible analysis. For evaluation of the models, evaluation parameters such as MAE (Mean Absolute Error), NMAE (Normalized Mean Absolute Error), RMSE (Root Mean Square Error), NRMSE (Normalized Root Mean Square Error), and evaluation metrics such as hit rate, false alarms rate, miss rate, and resource utilization are used. Out of the three, the transformer model emerged as the most suitable model for this job with very high precision and effectiveness and extremely low MAE as well as a high hit rate. The authors stated that future research will focus on hybrid multi-objective models and prioritize security & privacy issues in this field.

This study by Levitin et al. [23] proposes a probabilistic model that optimizes the uploading and downloading pace distribution between non-identical storage units, where both systems are used to maximize MSP. It advances the field by addressing upload/download rate and its connection to the failure rate of storage units. This methodology provides a framework for evaluating MSPs under system demand. The approach is demonstrated with different parameters among the same and different ranks. However, the solutions and models of this research aren't applied in the real world.

A study by Cui and Wang [37] suggests using EC-KAD, a scheme that uses erasure coding integration with Kademlia DHT protocol for optimization of object storage. The working order of this scheme is to use a Reed-Solomon code that is used for fault tolerance, and also uses the Kademlia protocol for efficiently locating data blocks. During tests on the Hadoop platform, the scheme shows signs of improved performance in retrieval and in overhead reduction compared to traditional replication. The system was tested in a controlled environment without addressing the performance in real-world heterogeneous data or in a global production cloud.

In a study by Xu et al. [12] evaluates incremental encoding to optimize new data encoding in erasure coded storage systems using a decentralized framework. The core proposal of this study is to encode data directly and incrementally across servers in a pipelined protocol while also avoiding the initial replication of other methods. This research also implemented Hadoop and cross-checked the results through the usage of single and cross-datacenter environments. Evaluations show a better trade-off compared to other models with 44-48% less network traffic, even with the same I/O performance as replication encoding, and still retaining superior read/write performance than strip encoding. This paper mainly focused on linear erasure codes and keeping degraded read operation optimization for future work.

This comprehensive research review by Saadoon et al. [21] strategically checks for fault tolerance and solutions in big data storage and processing systems. The paper

focuses on classifying fault types, analyzing detection and recovery protocols, and comparing their pros and cons with system performance. It works as a broad survey comparing know-how from various studies in an effort to explain the mechanisms for achieving reliability and availability in large-scale operations. But as a high-level review, this does not propose any novel technique. The study concludes by finding out the research gaps and proposing future paths to solve evolving fault tolerances in big data.

A paper by Muntala and Karri [27] shows a conceptual framework to manage end-to-end machine learning lifecycle with Oracle Cloud Infrastructure. Its main focus is on ERP-related applications. Its core attributes are to build an architectural guide to use in the integration of OCI data with Oracle Fusion ERP data to build an advanced predictive analytics to be used for expense forecasting. The proposed approach was to use Autonomous Databases and Object Storage to control the stages from data ingestion to model monitoring. But the paper does not include a real-world implementation with qualitative experimental results, nor does it include performance benchmarks for such environments. The research converges on future challenges in data drift and security, cementing it as the foundation for future research in ERP systems for operationalizing AI.

A research paper by Patil and Sharma [17] proposes a ML based algorithm for resource allocation in mobile and cloud computing environments. The novel approach used in this study was to use feature extraction and optimization algorithms for eliminating unused cache files to reduce data blocks and device storage overhead. The implementation was done in Python using tensorflow to work on real world data where it claimed to have freed up 2.5% memory leading to better processing speed. But this paper's lack of comparisons with other established benchmarks makes it slightly limited in scope. The paper concludes on a detailed analysis of machine learning's role in validation on its reported effectiveness.

## 2.3 Summary of Key Findings

The studies reviewed in this paper show that in most cases, the dominant research paper is based of erasure coding and its performance trade-offs, automation and intelligent control using ML/RL for storage management, system-level optimizations for EC write/update overhead and analytical and architectural approaches for predicting or reducing latency and operational overhead.

A key finding from EC-focused benchmarking shows that stronger fault tolerance has higher user-visible cost. Noor et al. [39] demonstrated that the more fragments and parity increase lead to longer upload, download, and delete times. This highlights that EC policy selection is a trade-off decision, not always optimal. Also, we can see that erasure-coding is more write and update efficient for storage. A study by Pei et al. shows that Group-U is a workload-aware grouping and pipelined transmission to reduce update cost. This reduces time and overhead compared to the baseline. Also, Xu et al. demonstrated that erasure coding reduces network traffic while maintaining I/O performance by avoiding initial replication [12].

From our second key finding, ML/RL is applied to decisions such as storage placement, load balancing, metadata optimization, and deduplication. A classification-based storage is shown by Mondal et al. [20], where this routing can reduce administrative effort and improve efficiency. Also, a study by Noel et al. [13] proposed an RL-based ARM system where automated control outperforms default storage behavior by dynamically reacting to hotspots in Ceph. Finally, researchers use analytical models to reduce latency and compute overhead. A study by Su et al. [10] apply queuing theory to predict tail latency percentiles for Swift. Multi-cloud placement optimization demonstrates that combining coding with placement heuristics can reduce cost and latency under QoS constraints.

Across the reviewed literature, we find that object storage optimization has been addressed through isolated techniques such as erasure coding design, system-level optimizations, analytical modeling, and ML/RL-based control for auxiliary storage decisions. Existing studies primarily evaluate fixed EC configurations or optimize specific subsystems. However, there is limited work that directly formulates EC policy selection itself as a data-driven, workload-aware decision problem. In particular, studies on ML that recommend optimal EC policy based on workload context and system behavior are largely unexplored, which motivates the need for our proposed approach.

Table 2.1 summarizes the key attributes of the reviewed studies for comparative analysis.

| Author name | Year Published | Used Algorithms / Architecture | Dataset / Test Environment | Sector of optimization | Use of Erasure Coding? |
|---|---|---|---|---|---|
| Noor et al. [39] | 2025 | Reed–Solomon (RS) codes (RS(5+3), RS(7+5), RS(10+4)); SimEDC simulator; OpenStack Swift. | Custom dataset (MCSD-100) and COCO-17 benchmark; local and remote testbeds. | Time efficiency & fault tolerance benchmarking for I/O performance in OSS. | Yes (primary focus) |

| Author name | Year Published | Used Algorithms / Architecture | Dataset / Test Environment | Sector of optimization | Use of Erasure Coding? |
|---|---|---|---|---|---|
| Pei et al. [8] | 2016 | Group-U: a three-layer framework with workload-aware grouping and pipelined transmission. | Implemented on Raid Distributed Storage System (RDFS); testbed experiments on physical/virtual machines. | Update efficiency for in-place updates in erasure-coded storage. | Yes (core problem context) |
| Levitin et al. [23] | 2023 | Probabilistic model using Proportional-Hazards Model (PHM) / Accelerated Failure-Time Model (AFTM). | Case study of a water supply system with two tanks (storage units). | Mission reliability and pace-dependent failure in production-storage systems. | No |
| Cui and Wang [37] | 2025 | EC-Kad: RS erasure codes with Cauchy matrix + Kademlia DHT protocol. | Implemented and tested on the Hadoop cloud storage platform. | Fault tolerance & retrieval efficiency in cloud storage. | Yes (core component) |
| Xu et al. [12] | 2018 | Incremental encoding: a decentralized, pipelined encoding framework for linear codes. | Implemented in Hadoop 3.0; evaluated in single and cross-datacenter environments (UCloud). | Encoding efficiency and network traffic for writes in geo-distributed storage. | Yes (core problem context) |
| Kamble et al. [32] | 2024 | LSTM, CNN, BERT. | Google Cluster Data. | Resource utilization, cost-effectiveness, and performance. | No |

| Author name | Year Published | Used Algorithms / Architecture | Dataset / Test Environment | Sector of optimization | Use of Erasure Coding? |
|---|---|---|---|---|---|
| Saadoon et al. [21] | 2021 | Survey/review: classifies fault types (permanent/transient) and tolerance approaches (detection, recovery). | N/A (review paper). | Fault tolerance in big data storage and processing systems. | Discussed as a key technique |
| Muntala, Karri [27] | 2023 | Conceptual framework using Oracle Cloud Infrastructure (OCI) Data Science and Autonomous Database. | Proposed for Oracle Fusion ERP data; expense forecasting case study. | ML lifecycle management for predictive analytics in ERP. | No |
| Patil, Sharma [17] | 2020 | Machine learning for feature extraction and optimization; implemented in Python with TensorFlow. | Real file system data and cache files to test memory optimization & dynamic resource management. | Memory management for cloud/mobile environments. | No |
| Su et al. [10] | 2017 | Queuing-theory performance model. | OpenStack Swift testbed. | Tail latency percentile prediction. | No |
| Matt et al. [9] | 2017 | Heuristic placement optimization of cost/latency-aware policies. | UbuntuOne-based multi-cloud pricing/latency measurement. | Multi-cloud placement to reduce cost & latency under QoS constraints. | Yes |
| Mondal et al. [20] | 2021 | Supervised classification for storage placement. | RSoS + Ganglia system metrics: healthcare-type data case. | Storage placement recommendation. | No |

| Author name | Year Published | Used Algorithms / Architecture | Dataset / Test Environment | Sector of optimization | Use of Erasure Coding? |
|---|---|---|---|---|---|
| Durner et al. [25] | 2023 | AnyBlob, object scheduler, and object storage architecture. | AWS experiments. | Storage cost optimization and high-performance data retrieval. | No |
| Su et al. [22] | 2021 | ARIMA, LSTM, DQN, PSO, linear regression. | DevOps, AWS, and Azure Effectiveness Deployment Dataset. | Private data sharing in edge-cloud collaboration. | Yes |
| Syed & Albalawi et al. [35] | 2024 | Predictive ML and optimization framing for resource allocation. | Green computing and DevOps; AWS and Azure effectiveness deployment datasets. | Resource management optimization. | No |
| Chen et al. [30] | 2024 | Hybrid metadata system for hot/cold metadata handling. | COSBench workload. | Metadata maintenance. | No |
| Noel et al. [13] | 2019 | RL-based adaptive resource management. | Ceph testbed. | Response latency requirement. | No |
| Akbar et al. [24] | 2023 | TTD+DPC chunking, multi-level hashing. | Cloud deduplication evaluation setup. | Dedup ratio + throughput + CPU overhead reduction. | No |
| Bucur et al. [11] | 2018 | Review of single-cloud and multi-cloud object storage systems. | N/A. | Multi-cloud object storage architecture. | No |
| Factor et al. [1] | 2005 | T10 OSD protocol framing. | Controller throughput discussion. | Object storage architecture foundation. | No |

| Author name | Year Published | Used Algorithms / Architecture | Dataset / Test Environment | Sector of optimization | Use of Erasure Coding? |
|---|---|---|---|---|---|
| Hu et al. [31] | 2024 | TVM-based EC library mapping. | TVM runtime. | Reduced developer complexity and faster EC implementation. | Yes (focused) |

Table 2.1: Table of Key Findings

# Chapter 3

# Requirements, Impacts and Constraints

## 3.1 Final Specifications and Requirements

### 3.1.1 Hardware Configuration

The dataset was created using a virtual machine with 8 GB of base memory and 12 CPU cores. The virtual machine was running on a computer with an Intel Core i5-12500H and 16 GB of RAM.

### 3.1.2 Software Configuration

The dataset was generated on a virtual machine with Ubuntu 18.04.6. Also, OpenStack Swift 2.23.3 (Train) [43] was used to create a storage cluster with erasure-coding capabilities. For training the ML models, we used Google Colab and Python libraries like numpy [16], sklearn [2], and pandas [18] were used.

## 3.2 Societal Impact

The implementation of a smart EC policy goes beyond technical terms, extending to positive social outcomes.

- **Democratising durable storage:** By reducing the storage cost on per unit of data storage, the framework makes large-scale data more accessible to resource strained part of the population
- **Reducing digital divide:** Lowering infrastructure cost would help to reduce the online classism between the underserved and the wealthy by bridging the gap with affordable cloud services.

## 3.3 Environmental Impact

The environment would be a direct and significant beneficiary of the improved storage efficiency when selecting the optimal EC policy.

- **Reduced E-waste:** Better storage utilization would lead to less consumption of storage and discarding of old memory, directly translating to a reduction in raw material extraction and eventual waste.
- **Lower energy consumption:** Less storage would require less energy, which leads to reduced energy consumption. The cooling of these huge data centers would also drop significantly, leading to less thermal pollution from both power plants and data centers.
- **Supporting green initiatives:** By lowering energy consumption and resource consumption, optimized object storage systems will better align with broader sustainability goals and support environmentally friendly and green computing initiatives.

## 3.4 Ethical Issues

The project used to predict system-level performance evaluation and make decisions based on the generated dataset; the ethical considerations in this were in regard to data integrity, transparency, fairness, bias, and reproducibility.

- **Data integrity as a public good:** For public or scientific data stored using EC, the ethical step is to ensure the data stays uncorrupted over the years.
- **Transparency:** Industry storage systems should clearly document how machine learning–based optimization influences data handling and policy selection. Since user data is used to analyze against the ML model, the user should be informed (via terms and services) about the ways in which their data is being used.
- **Fairness and bias:** Since the policy recommendations come from the ML model, the dataset used has to be diverse and not to overrepresent one type of data (e.g., read-only or write-only) to ensure fair and unbiased results.
- **Reproducibility:** Ethical research ensures that results and the dataset used in this study are reproducible by following the footsteps in the documentation to be easily recreated and tested.

## 3.5 Project Management Plan

This project follows structured and planned steps to ensure consistent documentation and smooth execution of the research.

- **Requirement understanding:** The initial stage of this study was to understand the OpenStack Swift architecture, object storage fundamentals, and the inner working of erasure codings. Also, key metrics such as storage overhead, latency, etc., are studied thoroughly.
- **Literature review:** Related papers and existing research on these topics are thoroughly studied to find research gaps and to understand the existing optimization approaches.

- **Dataset creation:** We created a dataset by executing 3,000 workload contexts across 10 EC profiles, resulting in 30,000 scenarios.
- **ML model development:** After identifying the patterns and trends in the dataset, regression and classification ML models are utilized as a viable approach to evaluate the dataset. XGBoost, Random Forest Regression and CatBoost were used for regression, and XGBoostClassifier, CatBoostClasifier and Logistic Regression were used for classification analysis.
- **Testing and validation:** The ABDS-30k dataset is validated using documentation-based invariant checks and repeatability experiments by reproducing a 120-sample dataset on a separate testbed and comparing distribution trends through histograms, CDFs, and median±IQR.
- **Documentation and report writing:** is the final stage in the research, where all the results, sources, cross-checks and validation are compiled along with the graphs to create a coherent and structured final report.

## 3.6 Risk Management

This research revolves around risks associated with dataset validity, system instability, and model reliability. To prevent these, all experiments were run strictly on a controlled SAIO environment with fixed EC policies and workload settings, with carefully logged results. The most risky step, failure injection, was done in a controlled and reversible manner to avoid corrupting the testbed. The study employed a comparison of both regression metrics (MAE, RMSE, $R^2$) and policy-selection metrics (Top-k accuracy, regret) to detect unstable recommendations. Data cleaning and sanity checks were performed on every model to bar inconsistent entries from corrupting the framework. Lastly, version control was used to reverse any damage if any were done during any of the automation and ML phases.

## 3.7 Economic Analysis

This research involves building and evaluating an erasure-coded (EC) object storage testbed. While erasure coding is widely adopted due to its storage efficiency compared to replication, the economic benefits of EC depend on selecting an optimal policy for a given workload. A study by Su et al. [10] showed that increased response latency in object storage systems directly impacts SLA compliance and operational cost, making performance-aware policy selection economically important. Also another study by Noor et al., [39] shows that increasing EC redundancy significantly increases upload, download, and deletion time in OpenStack Swift. According to Shen et al., [40] they observe that Google Colossus and Facebook f4, erasure coding can significantly reduce storage redundancy compared to replication acheiving 1.2× to 1.5× storage overhead. But this has a higher computational cost and recovery overhead. As a result, selecting a suboptimal EC policy can lead to hidden economic penalties in the form of increased CPU utilization, network bandwidth consumption, and potential SLA violations. In current industry practice, EC policies use empirical benchmarking and heuristic trade-offs, which may not be suitable

for workload-specific performance characteristics. By avoiding suboptimal policy selection, the proposed approach can reduce unnecessary operational overhead. Even if the storage overhead is higher than users' expectations, they can properly utilise storage and have an advantage in storage cost.

# Chapter 4

# Dataset

In this chapter, we discuss the design, generation method, and characteristics of the dataset developed for this thesis- ABDS-30k. For the construction of this dataset, thirty thousand workload variations are executed on OpenStack Swift as a storage system under multiple administratively assigned erasure coding policies and controlled failure injections. So, each of the samples in the dataset represents an independent and unique experiment of a workload.

## 4.1 Motivation and Objectives

In the current world, most object storage systems, such as OpenStack Swift, offer multiple erasure coding policies to store objects, which vary in storage efficiency, latency, and fault tolerance. But, the selection of an optimal erasure coding (EC) policy is complex and includes several characteristics based on workload, system, and recovery behaviour in case of failures. Existing literature in this sector mostly focuses on only a few workloads and some hand-picked scenarios, such as comparison and benchmarking between EC schemes and replication [39]. Some of the works focus on using public traces such as MSR Cambridge, Harvard NFS, as seen in the study done by Chan et al. [5], but those are not explicitly documented outcomes based on variable workloads per EC policy. Hence, we created ABDS-30k for capturing the real performance of an object storage system in varying workloads and configurations with failure injections to use it for machine learning and recommend optimal policies based on the target features.

## 4.2 System Context and Experimental Assumptions

We created a realistic, small-scale object storage cluster in OpenStack Swift running in Ubuntu 18.04.6 as a virtual machine. Swift Train version was used for creating a single-node Swift cluster. We also used 14 XFS-formatted virtual disks for the storage backend. For erasure coding, we used 10 erasure coding profiles, which are EC-2-2, EC-3-2, EC-4-2, EC-4-3, EC-5-3, EC-6-3, EC-7-3, EC-8-3, EC-8-4, and EC-10-4.

## 4.3 Data Collection Architecture and Workflow

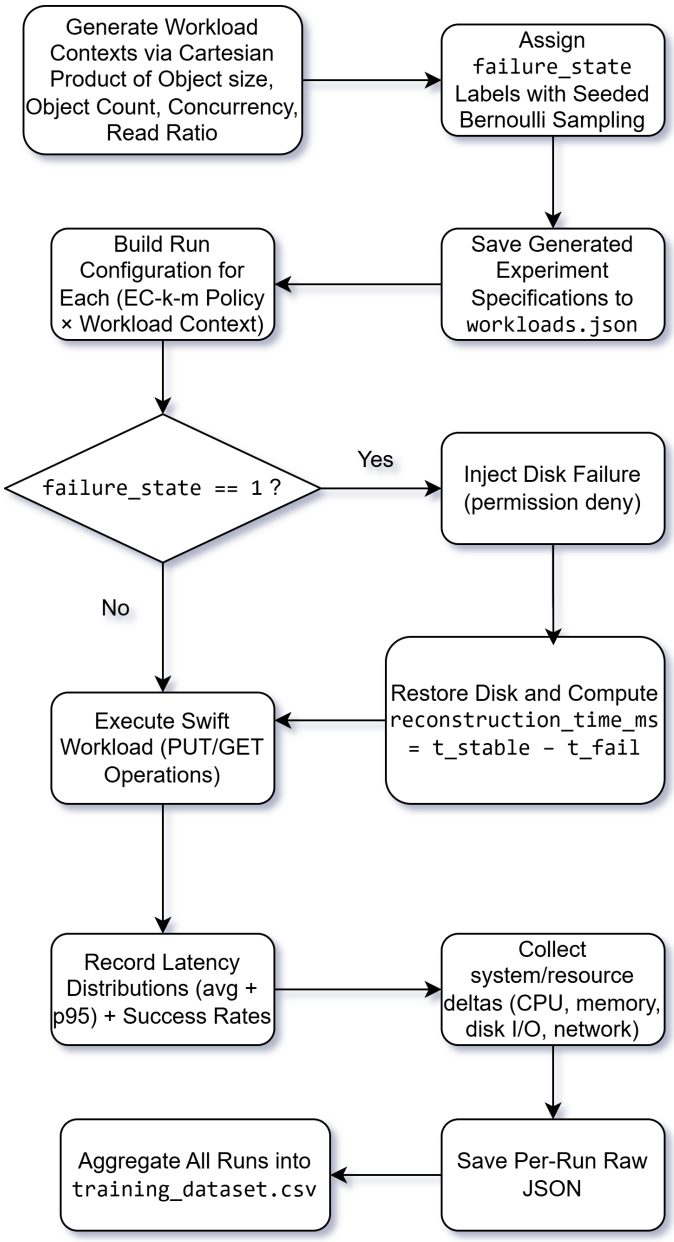


Figure 4.1: Dataset creation

Figure 4.1 shows our dataset creation process. The data collection process is designed to benchmark OpenStack Swift under different EC policies, workloads, and failure scenarios to generate a structured dataset. Here, architecture and workflow include the following components:

- Workload configurations contain file or object size, number of objects, concurrency level, and read/write ratio.
- Uploaded/downloaded data is generated by multiplying a randomized byte to create larger files to simulate real-world data.
- These workloads execute object upload and download operations using OpenStack Swift APIs and measure latency.

- Failure injection, which simulates disk failure by modifying disk permissions to emulate degraded storage conditions.

All workflows are executed using the Ubuntu terminal and Python automation tools.

Figure 4.2 to 4.7 shows the end-to-end reproducible SAIO-based dataset generation pipeline, including storage device provisioning, EC ring configuration, Swift service validation, policy-bound container setup, and a representative workload execution with controlled failure injection.

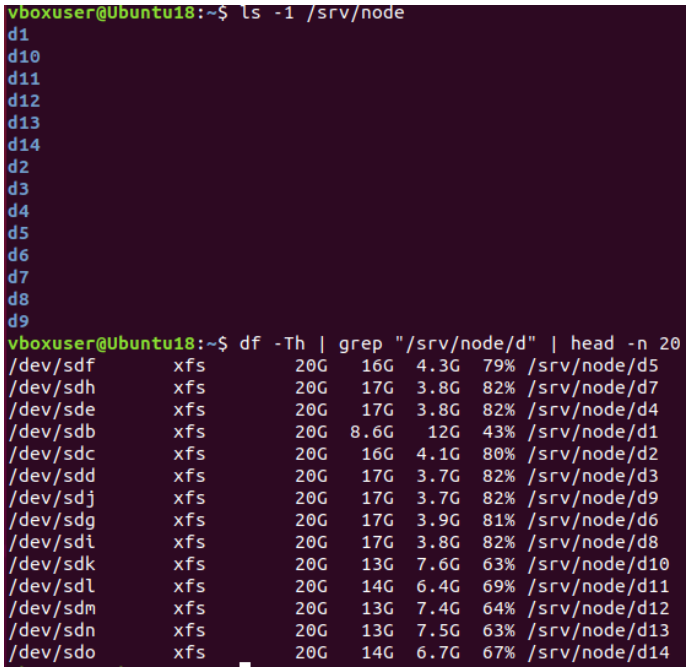


Figure 4.2: Mounted data disks for SAIO

```
vboxuser@Ubuntu18:~$ ls -1 /srv/node
d1
d10
d11
d12
d13
d14
d2
d3
d4
d5
d6
d7
d8
d9
vboxuser@Ubuntu18:~$ df -Th | grep "/srv/node/d" | head -n 20
/dev/sdf       xfs        20G   16G  4.3G  79% /srv/node/d5
/dev/sdh       xfs        20G   17G  3.8G  82% /srv/node/d7
/dev/sde       xfs        20G   17G  3.8G  82% /srv/node/d4
/dev/sdb       xfs        20G  8.6G   12G  43% /srv/node/d1
/dev/sdc       xfs        20G   16G  4.1G  80% /srv/node/d2
/dev/sdd       xfs        20G   17G  3.7G  82% /srv/node/d3
/dev/sdj       xfs        20G   17G  3.7G  82% /srv/node/d9
/dev/sdg       xfs        20G   17G  3.9G  81% /srv/node/d6
/dev/sdi       xfs        20G   17G  3.8G  82% /srv/node/d8
/dev/sdk       xfs        20G   13G  7.6G  63% /srv/node/d10
/dev/sdl       xfs        20G   14G  6.4G  69% /srv/node/d11
/dev/sdm       xfs        20G   13G  7.4G  64% /srv/node/d12
/dev/sdn       xfs        20G   13G  7.5G  63% /srv/node/d13
/dev/sdo       xfs        20G   14G  6.7G  67% /srv/node/d14
```

Figure 4.3: Swift ring artifacts created for base and EC policies

```
vboxuser@Ubuntu18:~$ source ~/swift-project/swift_env.sh
vboxuser@Ubuntu18:~$ swift stat
                     Account: AUTH_admin
                  Containers: 71
                     Objects: 259676
                       Bytes: 76781321133
 Containers in policy "ec-3-2": 6
    Objects in policy "ec-3-2": 29292
      Bytes in policy "ec-3-2": 7834378240
 Containers in policy "ec-8-3": 5
    Objects in policy "ec-8-3": 18016
      Bytes in policy "ec-8-3": 5269602304
Containers in policy "ec-10-4": 5
   Objects in policy "ec-10-4": 34225
     Bytes in policy "ec-10-4": 8693972992
 Containers in policy "ec-4-2": 6
    Objects in policy "ec-4-2": 18801
      Bytes in policy "ec-4-2": 7392968704
Containers in policy "default": 15
   Objects in policy "default": 594
     Bytes in policy "default": 2252348310
 Containers in policy "ec-7-3": 5
    Objects in policy "ec-7-3": 18044
      Bytes in policy "ec-7-3": 5311348736
 Containers in policy "ec-2-2": 8
    Objects in policy "ec-2-2": 67763
      Bytes in policy "ec-2-2": 16691298327
 Containers in policy "ec-4-3": 5
    Objects in policy "ec-4-3": 18235
      Bytes in policy "ec-4-3": 5499240448
 Containers in policy "ec-5-3": 5
    Objects in policy "ec-5-3": 18104
      Bytes in policy "ec-5-3": 5374263296
 Containers in policy "ec-6-3": 6
    Objects in policy "ec-6-3": 18604
      Bytes in policy "ec-6-3": 7198785536
 Containers in policy "ec-8-4": 5
    Objects in policy "ec-8-4": 17998
      Bytes in policy "ec-8-4": 5263114240
                X-Timestamp: 1768242886.55547
              Accept-Ranges: bytes
               Content-Type: application/json; charset=utf-8
                 X-Trans-Id: tx91a6c6e2ecd3455085dab-00697a732c
       X-Openstack-Request-Id: tx91a6c6e2ecd3455085dab-00697a732c
```

Figure 4.4: Swift proxy validation using swift stat

```
vboxuser@Ubuntu18:~$ source ~/swift-project/swift_env.sh
vboxuser@Ubuntu18:~$ for c in ds25k-ec22 ds25k-ec32 ds25k-ec42 ds25k-ec43 ds25k-ec53 ds25k-ec63 ds25k-ec73 ds25k-ec83 ds25k-ec84 ds25k-ec104; do
>   echo "==== $c ===="
>   swift stat "$c" | grep -i "X-Storage-Policy"
> done
==== ds25k-ec22 ====
      X-Storage-Policy: EC-2-2
==== ds25k-ec32 ====
      X-Storage-Policy: EC-3-2
==== ds25k-ec42 ====
      X-Storage-Policy: EC-4-2
==== ds25k-ec43 ====
      X-Storage-Policy: EC-4-3
==== ds25k-ec53 ====
      X-Storage-Policy: EC-5-3
==== ds25k-ec63 ====
      X-Storage-Policy: EC-6-3
==== ds25k-ec73 ====
      X-Storage-Policy: EC-7-3
==== ds25k-ec83 ====
      X-Storage-Policy: EC-8-3
==== ds25k-ec84 ====
      X-Storage-Policy: EC-8-4
==== ds25k-ec104 ====
      X-Storage-Policy: EC-10-4
```

Figure 4.5: Verification of policy-bound dataset containers

```
vboxuser@Ubuntu18: ~
File  Edit  View  Search  Terminal  Help
  GNU nano 2.9.3                        /etc/swift/swift.conf

[swift-hash]
swift_hash_path_suffix = RzUfDdu32L7J2ZBDYgsD6YI3Xle7hTVO8/oaQbpTbI8=
swift_hash_path_prefix = OZ1uQJNjJzTuFaM8X3v%fsJ1lR#FBwJjf9uhRlABevQ4

[storage-policy:0]
name = default
default = yes

[storage-policy:1]
name = EC-2-2
policy_type = erasure_coding
ec_type = liberasurecode_rs_vand
ec_num_data_fragments = 2
ec_num_parity_fragments = 2
ec_object_segment_size = 1048576

[storage-policy:2]
name = EC-4-2
policy_type = erasure_coding
ec_type = liberasurecode_rs_vand
ec_num_data_fragments = 4
ec_num_parity_fragments = 2
ec_object_segment_size = 1048576

[storage-policy:3]
name = EC-6-3
policy_type = erasure_coding
ec_type = liberasurecode_rs_vand
ec_num_data_fragments = 6
ec_num_parity_fragments = 3
ec_object_segment_size = 1048576

[storage-policy:4]
name = EC-3-2
policy_type = erasure_coding

^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text  ^J Justify  ^C Cur Pos  M-U Undo
^X Exit  ^R Read File  ^\ Replace  ^U Uncut Text  ^T To Spell  ^_ Go To Line  M-E Redo
```

Figure 4.6: EC policy definitions in /etc/swift/swift.conf

```
vboxuser@Ubuntu18:~$ cd ~/swift-project/phase2
vboxuser@Ubuntu18:~/swift-project/phase2$ rm -rf results && mkdir -p results
vboxuser@Ubuntu18:~/swift-project/phase2$ WLJSON="$PWD/configs/workloads_test_12.json" ./scripts/run_experiments.sh
Workloads: 12  ok=7  fail=5
Workloads: 12
Policies:  10
Total runs: 120
Failure disk: /srv/node/d1
Results dir: /home/vboxuser/swift-project/phase2/results/run_20260129_023850

[1/120] policy=EC-2-2 workload=W0002
[RUN] policy=EC-2-2 workload=W0002
/home/vboxuser/swift-project/phase2/results/run_20260129_023850/raw_W0002_EC-2-2_051edf0d.json
[2/120] policy=EC-2-2 workload=W0005
[RUN] policy=EC-2-2 workload=W0005
/home/vboxuser/swift-project/phase2/results/run_20260129_023850/raw_W0005_EC-2-2_dfdffe23.json
[3/120] policy=EC-2-2 workload=W0006
[RUN] policy=EC-2-2 workload=W0006
/home/vboxuser/swift-project/phase2/results/run_20260129_023850/raw_W0006_EC-2-2_bb14d1dd.json
[4/120] policy=EC-2-2 workload=W0007
[RUN] policy=EC-2-2 workload=W0007
/home/vboxuser/swift-project/phase2/results/run_20260129_023850/raw_W0007_EC-2-2_1acbf87d.json
[5/120] policy=EC-2-2 workload=W0008
[RUN] policy=EC-2-2 workload=W0008
/home/vboxuser/swift-project/phase2/results/run_20260129_023850/raw_W0008_EC-2-2_9a452310.json
[6/120] policy=EC-2-2 workload=W0010
[RUN] policy=EC-2-2 workload=W0010
/home/vboxuser/swift-project/phase2/results/run_20260129_023850/raw_W0010_EC-2-2_7c61acab.json
[7/120] policy=EC-2-2 workload=W0011
[RUN] policy=EC-2-2 workload=W0011
/home/vboxuser/swift-project/phase2/results/run_20260129_023850/raw_W0011_EC-2-2_56b8c8b5.json
[8/120] policy=EC-2-2 workload=W0001 FAIL
[RUN] policy=EC-2-2 workload=W0001 FAIL
[FAILURE] Simulating disk down (permission deny) on /srv/node/d1
/home/vboxuser/swift-project/phase2/results/run_20260129_023850/raw_W0001_EC-2-2_84e8fc0f.json
[FAILURE] Restoring disk access on /srv/node/d1
[9/120] policy=EC-2-2 workload=W0003 FAIL
[RUN] policy=EC-2-2 workload=W0003 FAIL
[FAILURE] Simulating disk down (permission deny) on /srv/node/d1
/home/vboxuser/swift-project/phase2/results/run_20260129_023850/raw_W0003_EC-2-2_b4ef02df.json
[FAILURE] Restoring disk access on /srv/node/d1
[10/120] policy=EC-2-2 workload=W0004 FAIL
[RUN] policy=EC-2-2 workload=W0004 FAIL
[FAILURE] Simulating disk down (permission deny) on /srv/node/d1
```

Figure 4.7: Dataset generation execution log

## 4.4 Workload Design and Scenario Space

In our generated 30,000 labeled samples, the dataset includes six primary input features describing the workload and system. Each workload takes EC policy identifier, file size, number of objects, concurrency level, read ratio, and failure state. Each workload is a combination of different parameters. Here we set our object size or file size from 16 KB to 1 MB, number of objects is between 1 and 10. Concurrency level is the number of parallel client threads, and read ratio is the fraction of objects that are read after upload.

## 4.5 Failure Injection and Recovery Modeling

Failure injection is modeled by temporarily simulating disk unavailability at the storage layer during workload execution. OpenStack Swift system detects failure when the selected storage device is made inaccessible using permission-based access denial. During failure, Swift object servers experience real I/O failure. It triggers EC fragment construction during reading. Swift avoids destructive disk removal or system crashes. In the workload, 40% of runs are executed under simulated failure conditions. So, each workload-policy pair is tagged with a failure state label or failure_state=1. This temporarily makes one storage disk unavailable, and the system is forced to operate in degraded mode. This triggers erasure coding recovery behavior. A recovery-oriented metric, reconstruction_time_ms, captures the time to access data under failure conditions.

## 4.6 Feature Engineering and Target Variables

### 4.6.1 Input, Output, and Learning Setup

Each line of the dataset represents an experiment run. The ML works here as a multi-target performance predictor with one regression model implemented per target metric. The features used as input features include system state, workload context, policy configuration, etc. The outputs are then measured in metrics that are also visible to the user.

**Targets (regression outputs):**

- avg_upload_ms
- p95_upload_ms
- avg_download_ms
- p95_download_ms
- reconstruction_time_ms

### 4.6.2 Candidate EC Policies

The recommendation steps a wide variety of EC policies (e.g, 4-2, 2-2, 6-3, etc) to find the most efficient one. These policies are written as EC-k-m, where k = data fragments and m= parity fragments. The policy storage properties are derived from these equations:

- **Storage overhead factor:** storage_overhead = $\frac{k+m}{k}$
- **Storage efficiency:** efficiency = $\frac{k}{k+m}$

These are used in the resulting score to determine the efficient policy.

## 4.7 Dataset Construction Workflow

For dataset creation process, a workload generator to generate 3,000 workload combinations while assigning failure state label to 40% of them and and save them in a json file. After that, we run the experiment and collect raw results in one json file per run. For failure states, we inject disk failure (permission deny). After all 30,000 runs are complete, we aggregate these results into a final csv file.

Below are the 5 scripts we used to generate the dataset:

**Algorithm 1:** Workload Specification Generation (`workloads.json`)

**Input :** Parameter grids: object sizes $S$, object counts $N$, concurrencies $C$, read ratios $R$; failure probability $p_f$; random seed *seed*

**Output:** `workloads.json`

SetRandomSeed(*seed*)
$workloads \leftarrow [\,]$
**foreach** $(s, n, c, r) \in S \times N \times C \times R$ **do**
  $w \leftarrow \{\}$
  $w$[`run_id`] $\leftarrow$ UniqueID()
  $w$[`object_size_kb`] $\leftarrow s$
  $w$[`num_objects`] $\leftarrow n$
  $w$[`concurrency`] $\leftarrow c$
  $w$[`read_ratio`] $\leftarrow r$
  $u \leftarrow$ `Uniform`$(0, 1)$
  **if** $u < p_f$ **then**
    $w$[`failure_state`] $\leftarrow 1$
  **else**
    $w$[`failure_state`] $\leftarrow 0$
  Append(*workloads*, $w$)
WriteJSON(`workloads.json`, *workloads*)

**Algorithm 2:** SAIO Dataset Run Orchestration (policy × workload)

**Input :** EC policies $P$; `workloads.json`; policy→container map; disk paths under `/srv/node/`; output dirs
**Output:** `results/raw/*.json`, `raw_runs.csv`

$workloads \leftarrow$ ReadJSON(`workloads.json`)
InitCSV(`raw_runs.csv`)
EnsureSwiftRunningAndReachable()
EnsurePolicyContainersExist()

**foreach** $policy \in P$ **do**
  $container \leftarrow$ ContainerFor($policy$)
  **foreach** $w \in workloads$ **do**
    $run\_id \leftarrow w$[`run_id`]
    $failure \leftarrow w$[`failure_state`]
    **if** $failure = 1$ **then**
      $disk \leftarrow$ SelectTargetDisk()
      $t_{fail} \leftarrow$ Now()
      FailureInject(DOWN, $disk$)
    $metrics \leftarrow$ RunWorkload($policy, container, w, run\_id$)
    **if** $failure = 1$ **then**
      FailureInject(UP, $disk$)
      $t_{stable} \leftarrow$ WaitUntilStableGETPUT($container$)
      $recon\_ms \leftarrow (t_{stable} - t_{fail})$
    **else**
      $recon\_ms \leftarrow 0$ `// or NA`
    $record \leftarrow$ MergeInputsOutputs($policy, container, w, metrics, recon\_ms$)
    WriteJSON(`results/raw/` $run\_id$ `.json`, $record$)
    AppendCSV(`raw_runs.csv`, $record$)

**Algorithm 3:** Permission-Based Failure Injection

**Input :** $action \in$ {`DOWN`, `UP`}; $disk\_path$ (e.g., `/srv/node/disk1`)
**Output:** Disk temporarily unavailable/available to Swift

**if** $action =$ `DOWN` **then**
  `chmod 000` $disk\_path$ `// deny access (simulate disk down)`
**else if** $action =$ `UP` **then**
  `chmod 755` $disk\_path$ `// restore access (simulate disk up)`

**Algorithm 4:** Workload Execution and Metric Collection

**Input :** Policy, container, workload $w$ (size, count, concurrency, read ratio), $run\_id$

**Output:** {`avg_upload_ms`, `p95_upload_ms`, `avg_download_ms`, `p95_download_ms`, `success_rate`}

$objs \leftarrow$ GenerateObjectNames($run\_id$, $w$[`num_objects`])
$U \leftarrow [\,]$; $D \leftarrow [\,]$
$u\_ok \leftarrow 0$; $d\_ok \leftarrow 0$

`// Upload phase (PUT)`
$obj \in objs$ with concurrency $w$[`concurrency`] $t_0 \leftarrow$ Now()
$status \leftarrow$ SwiftPUT($container, obj, w$[`object_size_kb`])
$t_1 \leftarrow$ Now()
Append($U$, $t_1 - t_0$)
**if** $status =$ ***SUCCESS*** **then**
  $u\_ok \leftarrow u\_ok + 1$

`// Download phase (GET)`
$reads \leftarrow$ SelectByRatio($objs$, $w$[`read_ratio`])
$obj \in reads$ with concurrency $w$[`concurrency`] $t_0 \leftarrow$ Now()
$status \leftarrow$ SwiftGET($container, obj$)
$t_1 \leftarrow$ Now()
Append($D$, $t_1 - t_0$)
**if** $status =$ ***SUCCESS*** **then**
  $d\_ok \leftarrow d\_ok + 1$

$avg\_up \leftarrow$ `mean`$(U)$; $\quad p95\_up \leftarrow$ `p95`$(U)$
$avg\_down \leftarrow$ `mean`$(D)$; $\quad p95\_down \leftarrow$ `p95`$(D)$
$succ \leftarrow (u\_ok + d\_ok)/(|objs| + |reads|)$
**return** $\{avg\_up, p95\_up, avg\_down, p95\_down, succ\}$

**Algorithm 5:** Dataset Aggregation (`results/raw` → `training_dataset.csv`)

**Input :** Raw per-run JSON files in `results/raw/`
**Output:** `training_dataset.csv`

$files \leftarrow$ ListFiles(`results/raw/*.json`)
$rows \leftarrow [\,]$
**foreach** $f \in files$ **do**
  $rec \leftarrow$ ReadJSON($f$)
  $row \leftarrow \{\}$
  `// Features`
  $row$[`policy_name`] $\leftarrow$ $rec$[`policy_name`]
  $row$[`failure_state`] $\leftarrow$ $rec$[`failure_state`]
  $row$[`object_size_kb`] $\leftarrow$ $rec$[`object_size_kb`]
  $row$[`num_objects`] $\leftarrow$ $rec$[`num_objects`]
  $row$[`concurrency`] $\leftarrow$ $rec$[`concurrency`]
  $row$[`read_ratio`] $\leftarrow$ $rec$[`read_ratio`]
  `// Targets`
  $row$[`avg_upload_ms`] $\leftarrow$ $rec$[`avg_upload_ms`]
  $row$[`p95_upload_ms`] $\leftarrow$ $rec$[`p95_upload_ms`]
  $row$[`avg_download_ms`] $\leftarrow$ $rec$[`avg_download_ms`]
  $row$[`p95_download_ms`] $\leftarrow$ $rec$[`p95_download_ms`]
  $row$[`reconstruction_time_ms`] $\leftarrow$ $rec$[`reconstruction_time_ms`]
  Append($rows$, $row$)
WriteCSV(`training_dataset.csv`, $rows$)

## 4.8 Dataset Statistics and Coverage Analysis

This section of the study provides a clear, comprehensive, and detailed overview of the dataset in the research to evaluate the performance of different erasure coding policies in OpenStack Swift. This dataset was generated in a controlled environment with simulated policy parameters, workload characteristics, and failure conditions. The dataset consists of 30,000 samples with 53 metrics that cover all of the EC configuration inputs and observed runtime outcomes.

The core objective of this dataset is to help in the findings of comparative policy analysis that are driven by data-driven policy selection with a keen eye on storage efficiency, fault tolerance, latency, throughput, and reconstruction cost.

### 4.8.1 Dataset Structure and Column Groups

The column metrics of the dataset can be classified into 5 major categories: policy parameters, workload characteristics, failure & degradation indicators, system resource metrics, and performance & I/O metrics. Each category gives a broader, detailed picture of what's happening in that respective sector.

**Policy Parameters**

- **policy_name:** Policy identifier (e.g., EC-6-3), representing a specific (k, m) combination.
- **Ec_k**: number of fragments (k) in which the data is divided.
- **Ec_m**: number parity (m) bit for reconstruction and redundancy.
- **Storage_overhead**: calculated as (k+m) /k for redundancy cost
- **Efficiency**: calculated as k/(k+m)

**Impact on policy:**

- More reliable, higher m means less chance of failure
- Higher m also means more storage cost
- Performance increases if a higher (k+m) is handled per operation.

In the experiment, these columns explain that higher redundancy may provide more durability, but also comes at increased encoding/decoding cost and reconstruction complexity.

**Workload Characteristics**

- **workload_id:** Identifier of the workload scenario/pattern.
- **File_size:** Object size
- **Num_object:** Number of objects
- **Concurrency:** Number of parallel client operations.
- **Read_ratio:** What fraction of operations were read (0.0 = write-only, 1.0 = read-only).
- **total_mb, read_mb, write_mb:** Total data breakdown of each operation
- **Io_intensity:** Indicator of workload pressure/strength.

**Impact on policy choice:**
Explains the workload for better policy choice:

- Read-heavy workloads work best with minimal fragmentation.
- Write-heavy data are more sensitive to encoding and overhang.
- Higher concurrency means increased contention and scalability differences between policies.

**Failure and Degradation Indicators**

- **failure_state:** Explains if a failure state happened
- **failed_disks:** Number of disk failures during the run.
- **failed_nodes:** Number of node failures
- **time_since_last_failure_s:** Time since last failure event.

**Impact on policy choice:**
These metrics evaluate the fault tolerance of the framework:

- Policies with higher m has less failure rate.
- Failure increases latency and lowers throughput due to reconstruction.

**System Resource Metrics**

These columns describe the system state before and during each workload execution:

- **data_disks_used_pct_before/after/delta:** Disk space utilization before and after the run.
- **cpu_util_pct_during_run:** CPU usage during workload execution.
- **mem_available_kb_before:** Available system memory before the run.
- **load1_before:** System load average prior to execution.
- **runnable_procs:** Number of runnable processes in the system.
- **dirty_kb, writeback_kb:** Indicators of memory write pressure.

**Impact on policy choice:**
Explains how system policies degrade performance under duress:

- Disk utilization and writeback pressure influence latency and throughput, especially for write-heavy workloads.
- Reconstruction time is a direct EC penalty metric and is critical when comparing policy robustness during failure conditions.

**Performance, Network, and I/O Metrics**

- **net_rx_bytes_delta, net_tx_bytes_delta:** Network receive/transmit traffic during the run.
- **disk_reads_delta_max, disk_writes_delta_max:** Peak disk I/O operations during the run.
- **disk_io_time_ms_delta_max, disk_weighted_io_time_delta_max:** Disk busy time and queue-weighted I/O time indicators.
- **disk_sectors_rw_delta_max:** Physical disk activity at the sector level.

- **avg_upload_ms, p95_upload_ms:** Average and tail (95th percentile) upload latency.
- **avg_download_ms, p95_download_ms:** Average and tail (95th percentile) download latency.
- **upload_success_rate, download_success_rate:** Operation reliability (success rate) under the given workload and failure condition.

**Impact on policy choice:**
These features are the driving factors behind selecting the "best" erasure coding policy:

- **Latency (especially p95):** Captures real-world user experience and stability under load.
- **Throughput:** Reflects bandwidth efficiency and scalability of the storage system.
- **Success rates:** Indicate the reliability of operations under workload and failure conditions.
- **Network usage:** Represents the network cost of fragment distribution and reconstruction traffic.

### 4.8.2 Dataset Trends

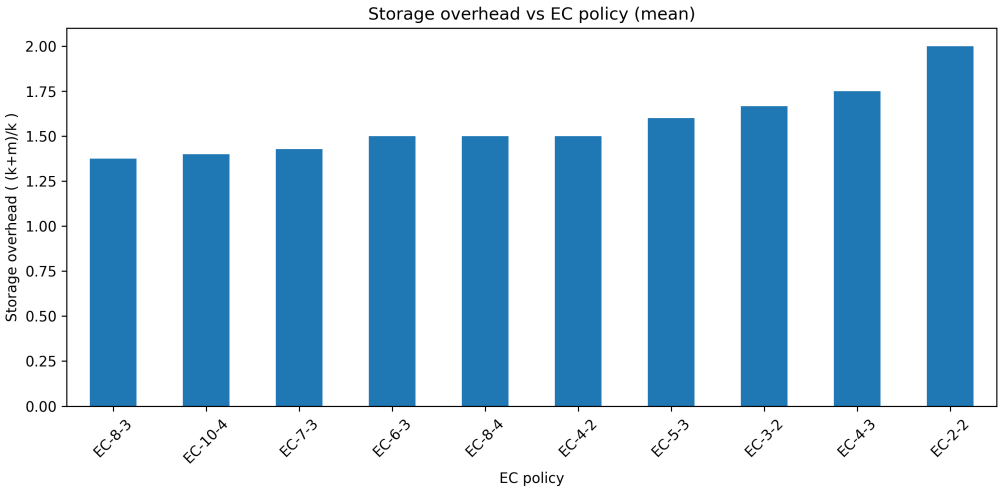


Figure 4.8: Storage overhead vs EC policy

Figure 4.8 illustrates how different EC policies vary in storage overhead, highlighting the trade-off between storage efficiency and fault tolerance across policy configurations.

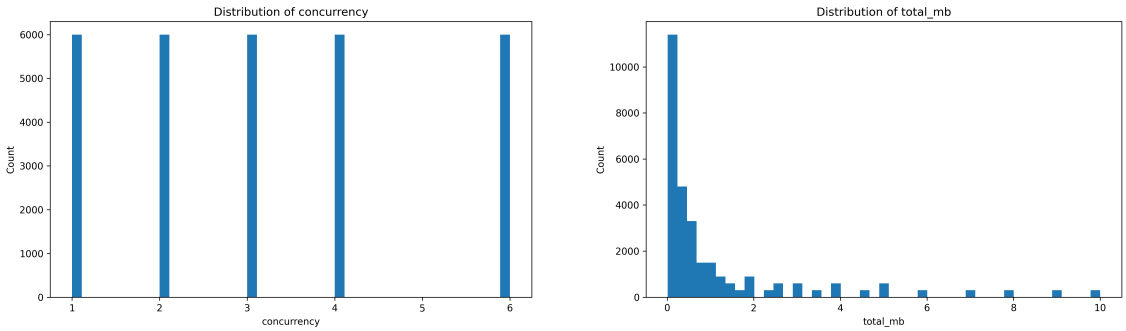


Figure 4.9: Distribution of concurrency and total mb within the dataset

Figure 4.9 demonstrates the diversity and coverage of workload intensity and parallel execution patterns represented in the dataset.

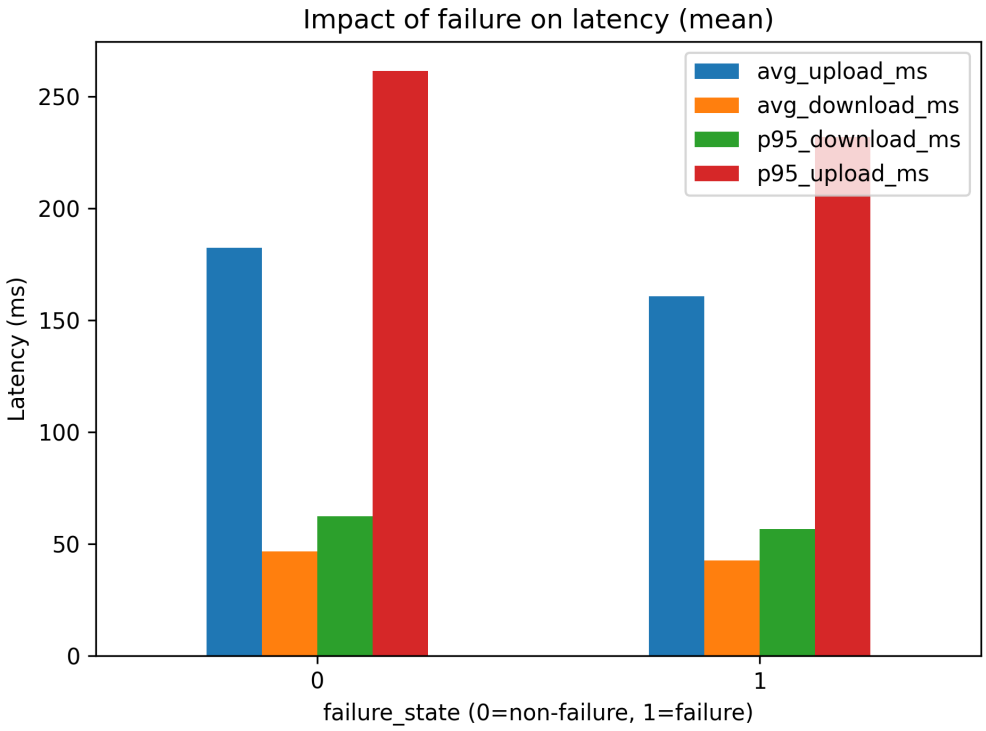


Figure 4.10: Impact of failure on latency

Figure 4.10 shows how failure conditions significantly increase latency due to reconstruction overhead and additional resource utilization.

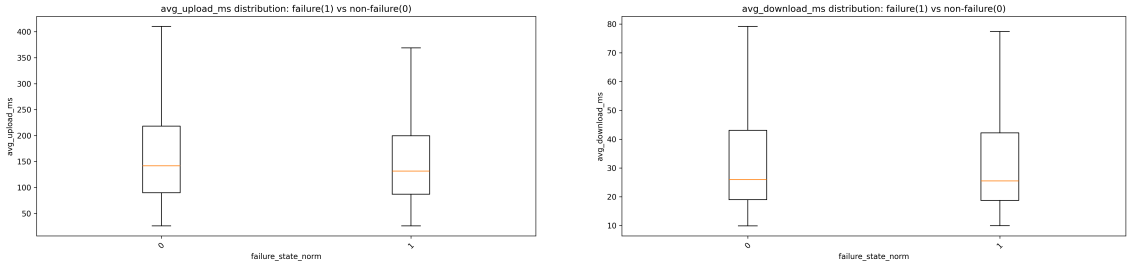


Figure 4.11: Average upload and download time distribution among failure and non-failure states

Figure 4.11 compares the variability and tail behavior of download latency between datasets, demonstrating consistency in performance spread and distribution trends.

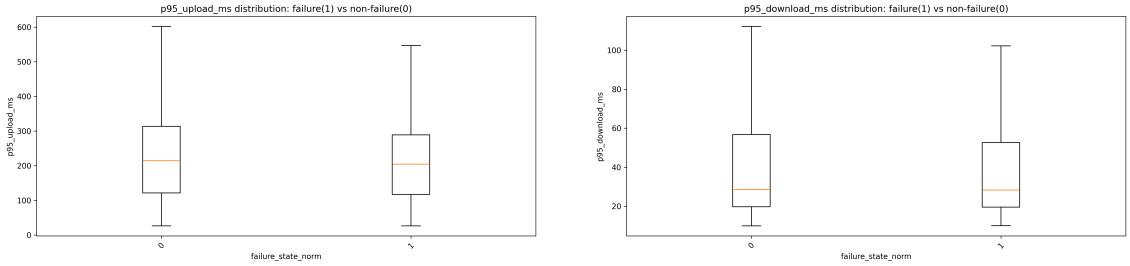


Figure 4.12: P95 upload and download time distribution among failure and non-failure states

Figure 4.12 highlights the similarity in upload latency patterns across datasets and failure states, confirming consistent workload impact and tail latency characteristics.

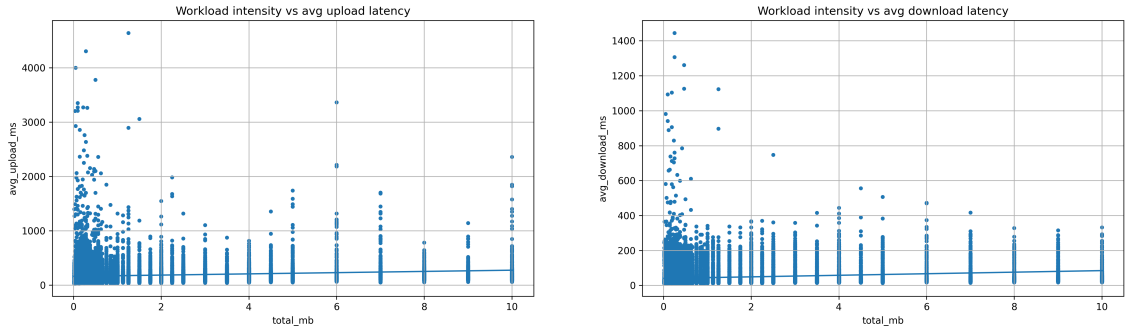


Figure 4.13: Comparison of workload intensity against average upload and download latency

Figure 4.13 illustrates that central tendency and variability of download and upload latency remain stable across datasets, reinforcing dataset reliability and representativeness."

### 4.8.3 Interconnections Between Metrics and Policy Choice

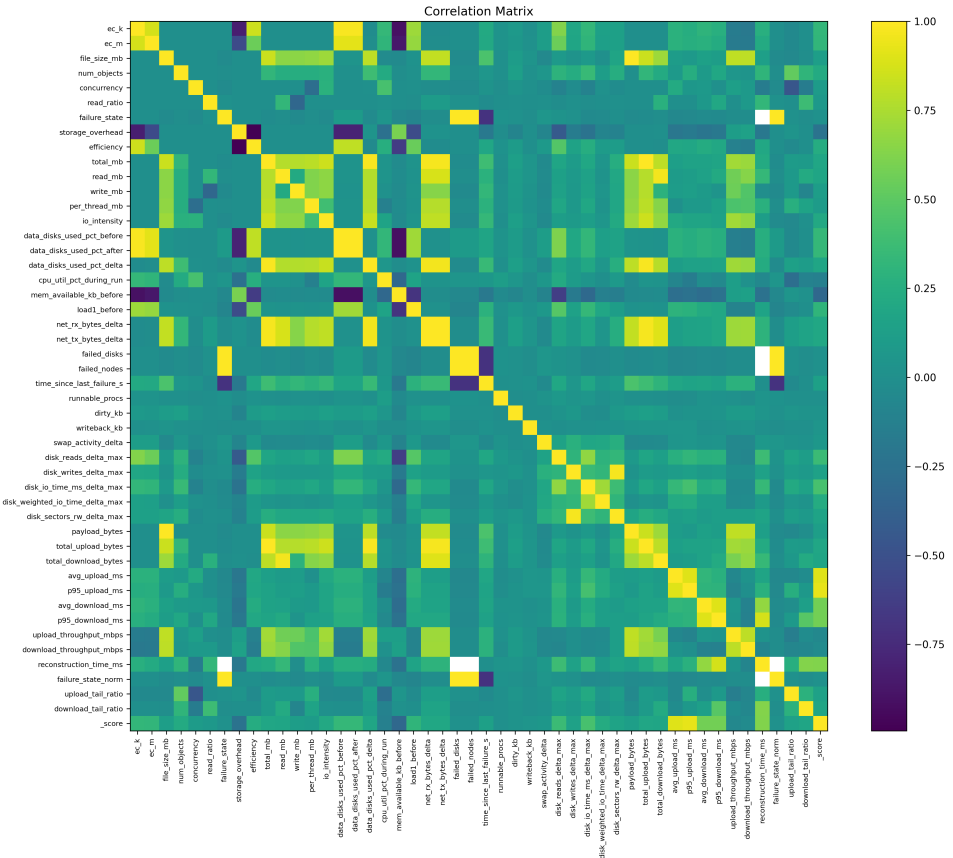


Figure 4.14: Correlation Matrix for ABDS-30k dataset

Beyond structural feature categorization, understanding inter-metric relationships is essential for EC policy decision modeling. For the interconnection analysis, we reduce the raw dataset from 53 features to 29 selected features, excluding metadata/identifier fields and redundant measurements that do not influence EC policy choice. The dataset analysis provides a deeper understanding of the key relationships that govern erasure-coding (EC) policy selection in OpenStack Swift.

1. **Efficiency vs. Fault Tolerance:** Increasing storage efficiency typically reduces fault tolerance, while stronger fault tolerance requires additional redundancy and higher storage overhead. Consequently, EC policy selection involves an inherent trade-off between cost efficiency and reliability.

2. **Workload-Dependent Behavior:** The performance of EC policies varies significantly with workload characteristics. Factors such as read_ratio, request concurrency, and overall workload intensity directly influence latency, throughput, and success rates, indicating that no single EC policy is optimal for all workload scenarios.

3. **Failure Impact and Reconstruction Cost:** Under failure conditions, reconstruction processes are activated, introducing additional disk I/O, network transfers, and computation. This leads to increased end-to-end latency and higher recovery overhead compared to failure-free operation.

4. **Resource Bottlenecks as Operational Constraints:** System-level limitations, including CPU utilization, disk bandwidth, and network capacity, constrain feasible EC policy choices. Policies with larger fragment counts may generate excessive traffic, potentially overwhelming host resources and resulting in degraded performance or increased operation failures.

### 4.8.4 Dataset Coverage and Limitations

The dataset provides an extensive coverage across 53 metrics for various EC configurations (varying k and m), workload patterns (read_ratio), concurrency levels, and both normal and failure-injected conditions for performance indicators and system bottleneck.

Also, the dataset is created in a controlled environment and may not fully represent the real-world data with production variations such as larger object sizes, multi-tenant behavior, and long-term background effects (e.g., rebalance).

### 4.8.5 Summary for Policy Selection

In short, the dataset supports a myriad of systematic EC policy evaluations using a combination of cost (storage_overhead, efficiency), performance (latency and throughput), reliability (success rates), repair impact (reconstruction_time_ms), and scalability signals (concurrency, resource trends). Optimal EC policy selection requires trade off between redundancy, efficiency, and performance stability based on deployment data.

## 4.9 Dataset Validation

To the best of our knowledge, no public dataset provides system metrics measurements for object storage systems in controlled workload and failure injection scenarios. Hence, we validate our dataset by mechanical correctness and invariant checks as per the official Object Storage System (OpenStack, Ceph) documentation and repeatability experiments in a separate testbed.

In ABDS-30k, the trends showcasing degraded read and reconstruction behaviour due to failure injection is supported by the behaviour reported in OpenStack Docs [42]. Also, the storage policy documentation from OpenStack states how different policies require different object rings which validates our experimental configuration of policy-specific containers and rings [41]. Ceph's EC plugin benchmarks also support the variance in the values as per k/m and configuration noticed in our dataset [4].

For further validation and repeatability check, we reproduced a 120-sample version of the same dataset in a different testbed with 120 workload contexts and 3 policies and the noticed value trends were the same.

The following histograms (Figure 4.9) highlight the frequency distribution of reconstruction time for failure rows and how the outliers occur across the two datasets,

where we notice a similar trend-

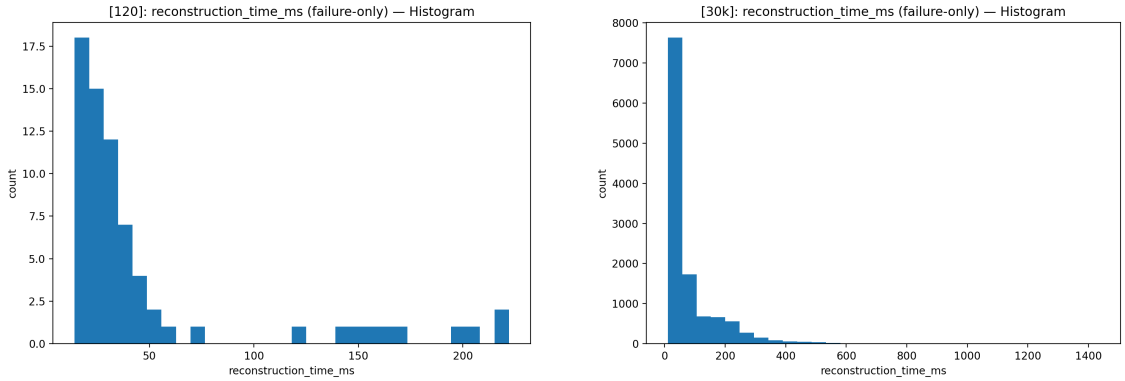


Figure 4.15: Histogram for frequency distribution of reconstruction time for failure rows

The following CDFs (Figure 4.10) highlight the failure cases that finish reconstruction under a specific amount of time for both of the datasets, where we also notice identical trends in metric distributions.

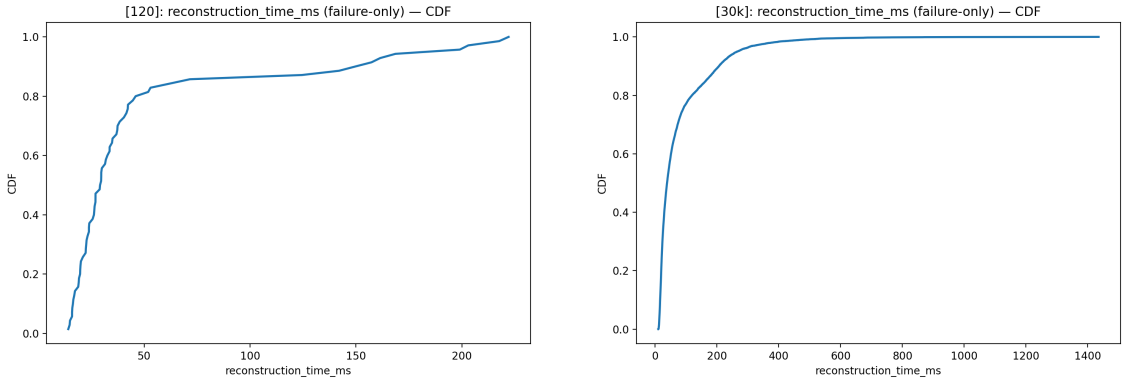


Figure 4.16: CDF for frequency distribution of reconstruction time for failure rows

The following graphs (Figure 4.11, 4.12, 4.13, 4.14) showcase policy median±IQR (Interquartile Range), where IQR = Q3-Q1, is a measure of spread with Q1 is 25th percentile, and Q3 is 75th percentile- so the graphs show the variability by ignoring the outliers. Here we can see identical trends for both of the datasets, which validates the data variability.

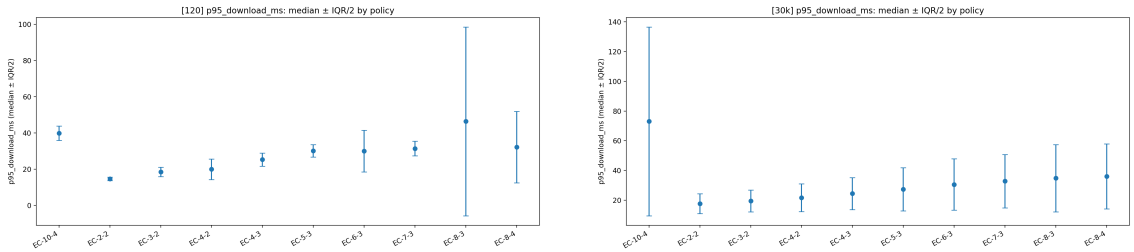


Figure 4.17: Comparison of median, IQR, and p95 download latency trends between the validation dataset (120 samples) and the main dataset (30k samples).

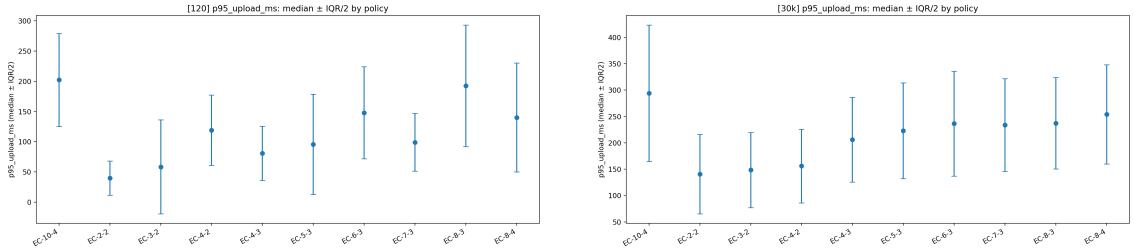


Figure 4.18: Comparison of median, p95 upload latency trends between the validation dataset (120 samples) and the main dataset (30k samples).

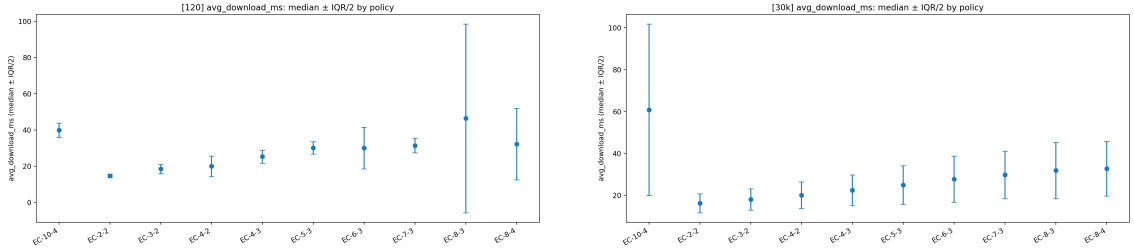


Figure 4.19: Comparison of median, IQR, and avg download latency trends between the validation dataset (120 samples) and the main dataset (30k samples).

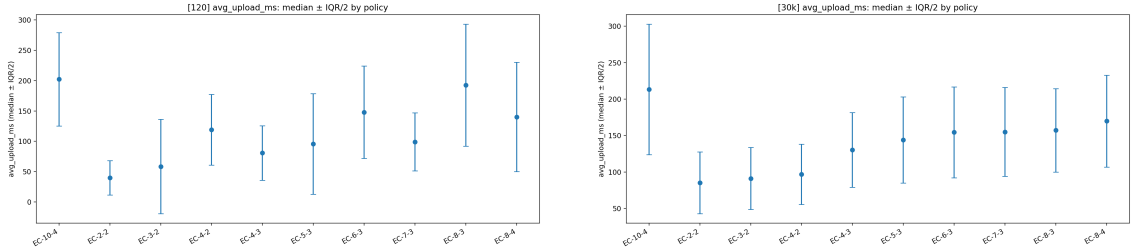


Figure 4.20: Comparison of median, avg upload latency trends between the validation dataset (120 samples) and the main dataset (30k samples).

## 4.10 Dataset Format and Availability

The dataset is available in csv file format, containing 30,000 rows and 53 columns. The dataset will be made publicly available for researchers and interested individuals to download.

## 4.11 Limitations and Intended Usage

This study presents a policy framework to propose EC policies for improved efficiency in object Storage systems. But in the real world, its scopes and practical uses are restricted by specific limitations, with further scope to work on in the future.

### 4.11.1 Limitations

The proposed framework does run into some performance and policy limitations:

- Performance trade-off: This framework prioritizes the storage overhang and its consequent latency over performance. But the performance-critical "hot" data is not considered in this study.
- Operational load during repair: Ec fragment reconstruction requires significant computational resources. Future work on this can be done using the Group U scheme or similar protocols to reduce these resource strains.
- Scale of evaluation: Testing was done only in a developmental scale and not on a real-world dynamic, corrupted, or unfinished system requiring petabytes of data to be fully be production ready.

### 4.11.2 Intended Use

This framework is intended to use upon automated tiered storage systems.

- Managing backup, archive, and server with a read-once, read-rarely pattern of data.
- Serving as the policy engine for administrators to reduce their workload.

# Chapter 5

# Methodology

## 5.1 Design Process and Methodology Overview

This research investigates how erasure-coding (EC) policy selection can be automated using machine learning based on workload context, using OpenStack Swift as a testbed. To test the feasibility of EC policy recommendation, we generated a dataset of 30,000 execution records, covering 3,000 distinct workload contexts, where each context is executed under 10 candidate EC policies. The goal of the model is to recommend the EC policy expected to minimize the performance cost given a workload context. For this, we use two distinctive machine learning paradigm- regression and classification.

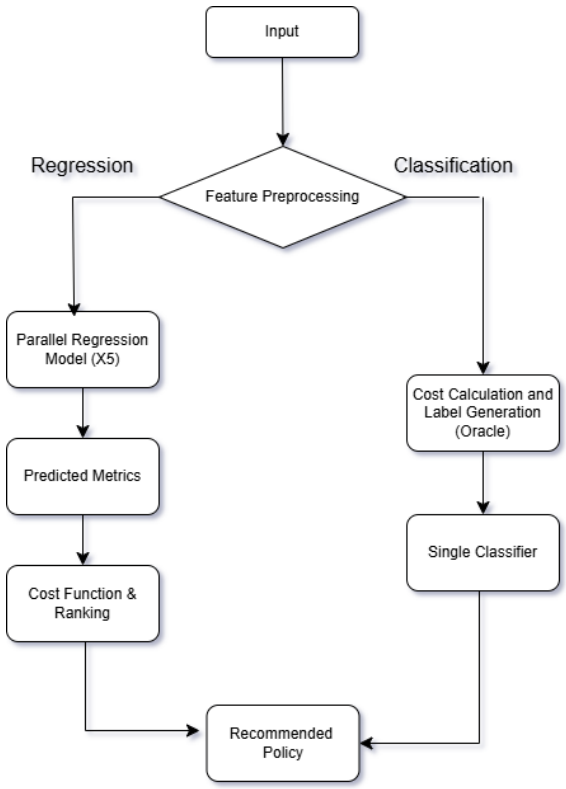


Figure 5.1: Proposed System Architecture

Figure 5.1 shows our full proposed system architecture. The first method uses

regression, where three separate regression models- CatBoost, XGBoost and RandomForest Regression are used for each of the target metrics, and the target features are predicted given a specific workload condition and EC configurations, and then scored based on a scoring function for recommending the best policy.

The second method formulates the problem as a classification task by computing a weighted score from measured latency and recovery metrics and by selecting the policy with the smallest score, a ground-truth label is created. Three classifiers- CatBoostClassifier, XGBoostClassifier and Logistic Regression were used for this method.

## 5.2 Model Design Specification

### 5.2.1 Feature Engineering

The models use a feature set for learning that includes several types of metrics such as

- policy_name, failure_disk are categorical.
- file_size_mb, num_objects, concurrency, read_ratio, failure_state are basic workload parameters.
- total_mb, read_mb, write_mb, per_thread_mb, io_intensity, payload_bytes are derived workload parameters.
- data_disks_used_pct_before, mem_available_kb_before, load1_before, cpu_steal_pct_before, runnable_procs, dirty_kb, writeback_kb, swap_activity_delta are indicators of pre-run system state.
- disk_reads_delta_max, disk_writes_delta_max, disk_io_time_ms_delta_max, disk_weighted_io_time_ms_delta_max, disk_sectors_rw_delta_max are disk pressure indicators.
- failed_disks, failed_nodes, failure_disk, time_since_last_failure_s are used for failure situations.

The target features are:

- `avg_upload_ms`, `p95_upload_ms` which indicate upload latency.
- `avg_download_ms`, `p95_download_ms` for download latency.
- `reconstruction_time_ms` for failure recovery performance.

### 5.2.2 Regression-Based Design Specification

In this method, we train three regression-based machine learning models- CatBoost, XGBoost, and RandomForest Regressor on our dataset to predict a total of five target features. After training, the model is passed on the demo workload with the available EC profiles and is used to suggest an optimal EC policy based on the score

that indicates weighted costs from the predicted values of the target metrics.

The whole workflow is done in four stages-

1. Dataset generation in a Swift All-In-One (SAIO) testbed by executing controlled workloads with injected failures under multiple erasure coding (EC) policies.
2. Selection of pre-run features that avoid leakage (i.e., only using measurements available before the workload executes).
3. Training one regression model per target metric: avg_upload_ms, avg_download_ms, p95_upload_ms, p95_download_ms, and reconstruction_time_ms.
4. Policy selection evaluation and recommendation based on the predicted performance for the given workload.

For policy recommendation, a normalized weighted score was defined with the values predicted of *avg_upload_ms*, *p95_upload_ms*, *avg_download_ms*, *p95_download_ms* and *reconstruction_time_ms* with separate values of weight = $w$ for each of them. The weights and normalization factors are represented as such–

- *avg_upload_ms* ($w = 1.0$, normalized by 138.42)
- *p95_upload_ms* ($w = 2.0$, normalized by 208.37)
- *avg_download_ms* ($w = 1.0$, normalized by 25.87)
- *p95_download_ms* ($w = 2.0$, normalized by 28.33)
- *reconstruction_time_ms* ($w = 1.5$, normalized by 37.83; only counted when *failure_state*=1)
- *storage_overhead* ($w = 2.0$, normalized by 2.0; computed deterministically from EC-$k$-$m$ as $(k + m)/k$)

$$\begin{aligned}\text{Score} = w_1 \frac{\text{avg\_upload\_ms}}{M_{\text{avg,up}}} + w_2 \frac{\text{p95\_upload\_ms}}{M_{\text{p95,up}}} + w_3 \frac{\text{avg\_download\_ms}}{M_{\text{avg,down}}} \\ + w_4 \frac{\text{p95\_download\_ms}}{M_{\text{p95,down}}} + w_5 \frac{\text{reconstruction\_time\_ms}}{M_{\text{recon}}} + w_6 \frac{\text{storage\_overhead}}{M_{\text{overhead}}}.\end{aligned} \tag{5.1}$$

where:

- $w_i$ are user-defined weights controlling the relative importance of each term,
- $M_i$ are normalization constants (medians of the corresponding metrics computed on the test set),
- reconstruction_time_ms is included only when failure_state = 1,
- storage_overhead is computed deterministically from the EC parameters as $(k + m)/k$,
- The score is dimensionless, and lower values indicate better policies.

### 5.2.3 Classification-Based Design Specification

The proposed design uses a classifier-only policy recommendation method, where the machine learning model predicts the best EC policy directly. For each workload context, each policy run has 5 measured targets, which are: average upload time, average download time, p95 upload time, p95 download time and reconstruction time. To define policy quality, a cost score is computed for each context:

$$S_{\text{total}} = W_{\text{perf}} \cdot S_{\text{perf,norm}} + W_{\text{storage}} \cdot S_{\text{storage}} \tag{5.2}$$

Where:

- $S_{\text{perf,norm}}$ is performance cost normalized within each context across policies.
- $S_{\text{storage}}$ is the storage overhead penalty scaled by workload volume.
- $W_{\text{perf}}, W_{\text{storage}}$ are fixed manually.

Performance cost uses read/write weighting and reliability for p95:

$$\begin{aligned} S_{\text{perf}} =& w\,(\text{avg\_upload} + 2\rho \cdot \text{p95\_upload}) + r\,(\text{avg\_download} + 2\rho \cdot \text{p95\_download}) \\ &+ 1.5 \cdot \text{recon} \end{aligned} \tag{5.3}$$

Where:

- r = read_ratio, w = 1-r
- Reliability factor for p95:

$$\rho = \min\left(1,\ \frac{\text{num\_objects}}{T}\right) \tag{5.4}$$

Here, T = Threshold

The weights used for score calculation are determined based on metric importance:

$$W_{\text{perf}} = 0.65, \quad W_{\text{storage}} = 0.35$$

For each workload_id, performance score is normalized within the range of 0 and 1:

$$S_{\text{perf,norm}}(c,p) = \frac{S_{\text{perf}} - \min_p S_{\text{perf}}}{\max_p S_{\text{perf}} - \min_p S_{\text{perf}} + \varepsilon} \tag{5.5}$$

Storage cost is derived from EC overhead and normalized across candidate policies, then multiplied by normalized workload volume:

$$OH = \frac{k+m}{k} \tag{5.6}$$

Then we normalize OH across candidate policies:

$$OH_{\text{norm}} = \frac{OH - \min(OH)}{\max(OH) - \min(OH)} \tag{5.7}$$

Then we compute workload volume proxy:

$$V = \log\left(1 + \text{write\_mb}\right), \qquad V_{\text{norm}} = \frac{V - \min(V)}{\max(V) - \min(V)} \tag{5.8}$$

Finally storage penalty is computed for all workload contexts:

$$S_{\text{storage}} = OH_{\text{norm}} \cdot V_{\text{norm}} \tag{5.9}$$

We used three classifiers to train and evaluate the models, which are CatBoostClassifier, XGBoostClassifier and Logistic Regression.

## 5.3 Data Preprocessing

### 5.3.1 Data Cleaning

To ensure consistency, valid learning, and prevent leakage, several columns are dropped- such as run_id, timestamp which are identifiers. ec_k, ec_m, storage_overhead, and efficiency are dropped as redundant, as these can be derived from policy_name. Columns such as cpu_util_pct_during_run, data_disks_used_pct_after, and data_disks_used_pct_delta are dropped to avoid leaky signals. The missing values in time_since_last_failure are ignored as CatBoost can handle NaN values.

### 5.3.2 Data Transformation

For CatBoost & XGBoost, log target transformation was used in training (log1p target transformation) to reduce the effect of extreme values and ensure stabilization in training. The model learns

$$y' = \log(1 + y) \tag{5.10}$$

and predictions are inverse-transformed back into milliseconds as:

$$y = \exp(y') - 1 \tag{5.11}$$

For classifier-only implementation, a new categorical feature was introduced for object-count binning. This helps the model in capturing shifts between very small and very large workloads.

### 5.3.3 Data Integration

Data reduction is done based on two approaches-

- Feature level reduction: Keeping only pre-run, decision-relevant data and excluding constant values
- Reconstruction time: For reconstruction time, only rows with failure state=1 are considered to avoid polluting the learning

### 5.3.4 Summary of Preprocessed Data

The dataset, after preprocessing, contains a total of 30,000 rows, out of which 12000 are failure rows (40%), and the feature set consists of a total of 29 columns, and the target set consists of 5 columns.

## 5.4 Implementation of Selected Design

The implementation workflow is script-based as of now and consists of 5 phases for the regression-based paradigm -

**Full pipeline:**
Preprocessing → Model training → Model evaluation → Policy selection evaluation → Recommendation

(a) Regression-only system.

**Classification-only system:**
Preprocessing → Score calculation → Model training → Model evaluation → Policy selection evaluation

(b) Classification-only system.

Figure 5.2: Pipeline flowcharts for the two approaches.

# Chapter 6

# Result Analysis

## 6.1 Performance Evaluation

### 6.1.1 Visualization

Figure 6.1, 6.2, 6.3 show the predicted vs actual scatter plots for each of the target features for CatBoost-

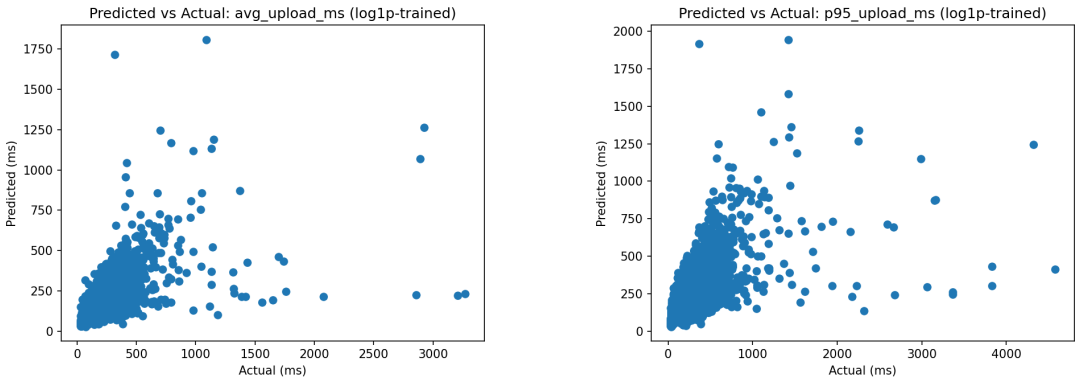


Figure 6.1: Predicted vs actual upload latency (CatBoost)

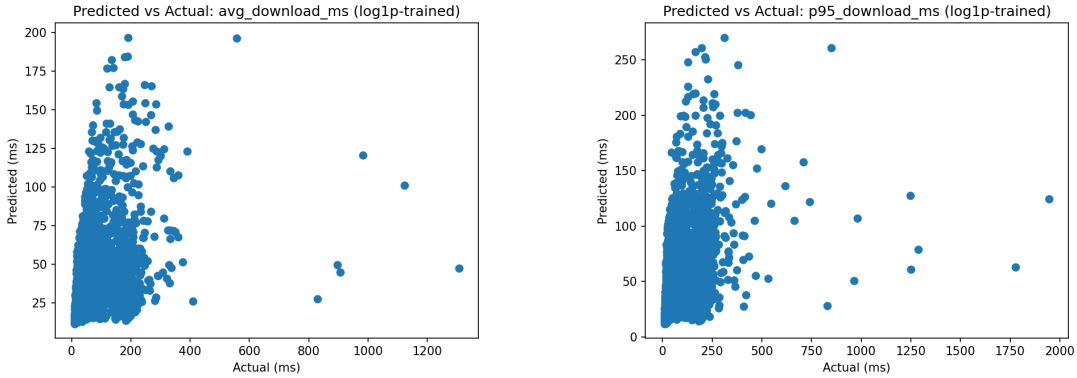


Figure 6.2: Predicted vs actual download latency (CatBoost)

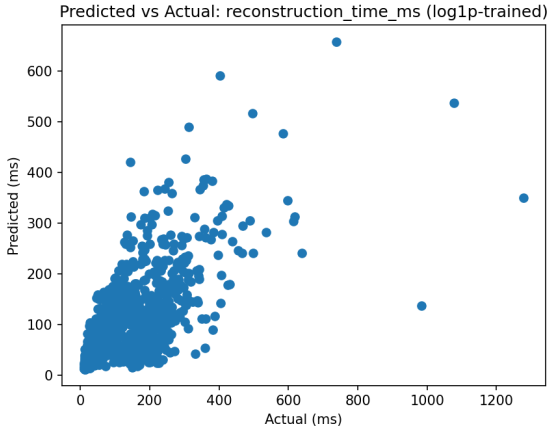


Figure 6.3: Predicted vs actual reconstruction time under failure (CatBoost)

We can see that for upload and download features- the overall trend is captured but in case of high latency values, the dispersion increases, which is explained by the nature of latency distribution in the system and the varying workloads. Also, the ordering of the policies stays optimal even with moderate regression error which favors policy selection and recommendation. Compared to other metrics, the reconstruction time plot shows stronger correlation due to stronger relations to failure conditions, recovery and disk behaviour.

The histogram in Figure 6.4 shows the regret distribution across varying workloads-

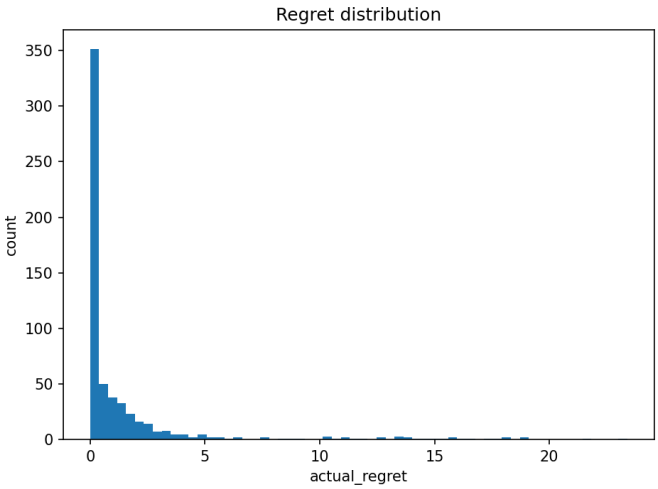


Figure 6.4: Regret histogram for upload latency prediction (CatBoost)

This plot shows the distribution of true regret values which is defined by getting the difference between the score of the recommended policy and the true best-performing policy. The plot being right-skewed indicates that in most of the cases the recommended policy is close to the optimal choice, and despite the model not always predicting the single optimal option, the severely worse options are avoided.

Figure 6.5 shows the regret CDF or cumulative distribution function, which showcases the reliability of the whole recommendation policy.

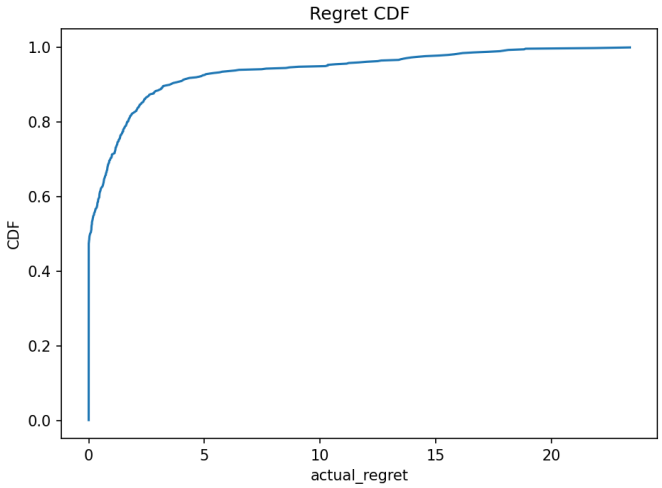


Figure 6.5: Regret CDF for upload latency prediction (CatBoost)

This shows that a large portion of the workloads have low regret values which means that even when the model does not select the exact optimal policy, it recommends a policy close to the optimal in almost all of the cases. This is further supported by the Top-1 accuracy of 47.17% and the Top-3 accuracy of 81.17%.

Similarly, for XGBoost, Figure 6.6, 6.7, 6.8 show the predicted vs actual scatter plots for each of the target features-

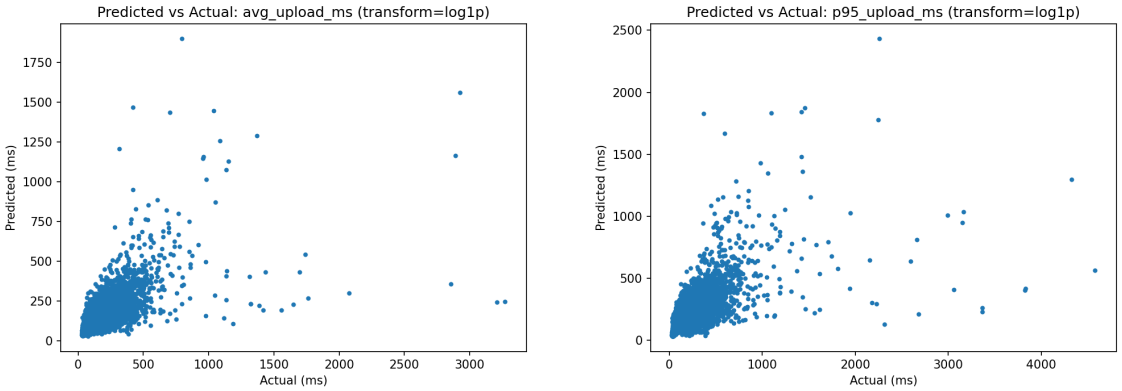


Figure 6.6: Predicted vs actual upload latency (XGBoost)

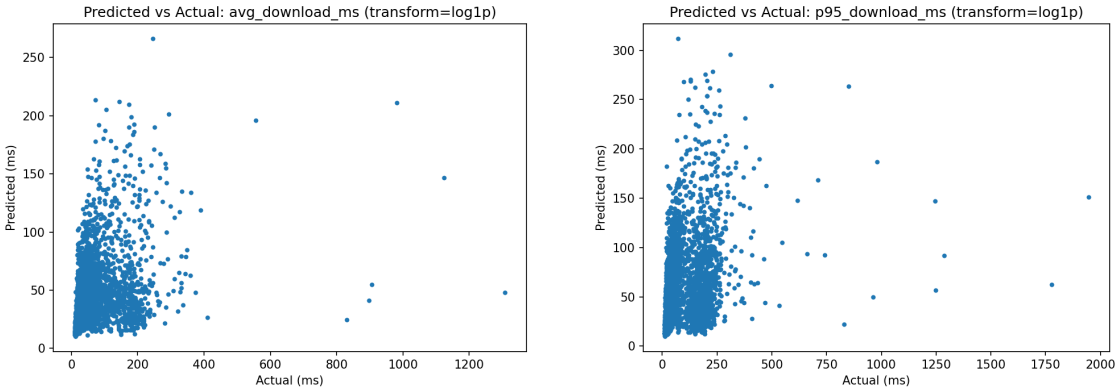


Figure 6.7: Predicted vs actual download latency (XGBoost)

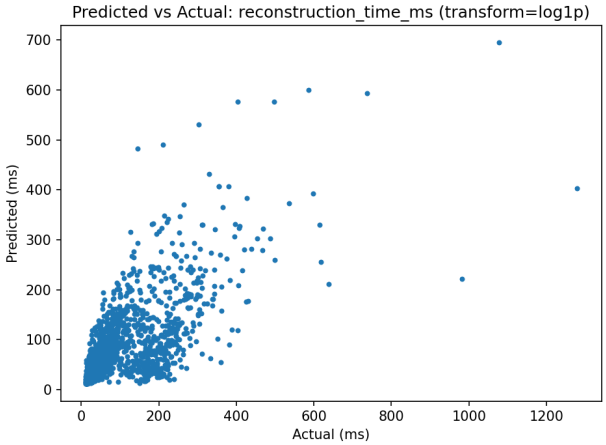


Figure 6.8: Predicted vs actual reconstruction time under failure (XGBoost)

To understand the prediction quality for the XGBoost model, predicted vs actual scatter plots were generated for each metric. The reconstruction time plot shows that the model captures the relationship between failure-related features and recovery latency. While many plots have larger reconstruction times, the ordering of sample is preserved. This is critical for policy rankings. For both average and 95th percentile upload latency, both prediction and actual values aligned well for the low-mid latency range. But in slow cases, alignments are messy or larger errors at extremes due to heavy-tail latency. In the case of avg_download and 95th percentile, we can see that the plot exhibits higher scatter compared to the upload metrics. This indicates that worst-case download delays are hardest to predict. As we can see in a few plots, some predicted values have huge differences with actual metrics.

The regret histogram and regret CDF for XGBoost are shown in Figure 6.9-

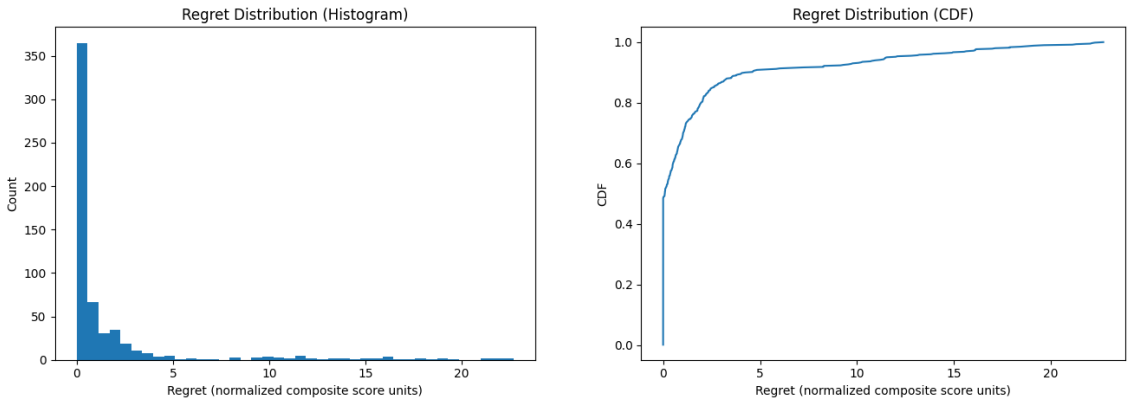


Figure 6.9: Regret histogram for upload latency prediction and Regret CDF for upload latency prediction

The regret histogram shows the full distribution of regret values. Many recommendations are near 0, which means they are very close to optimal. The CDF shows what fraction of contexts achieves regret below a threshold value. Values near 0 mean most workloads have low regret. Even though top-1 accuracy is not perfect, the regret distribution tells us that wrong choices are still near optimal.

As for Random Forest Regression, Figure 6.10, 6.11 and 6.12 show the predicted vs actual scatter plots for each of the target features-

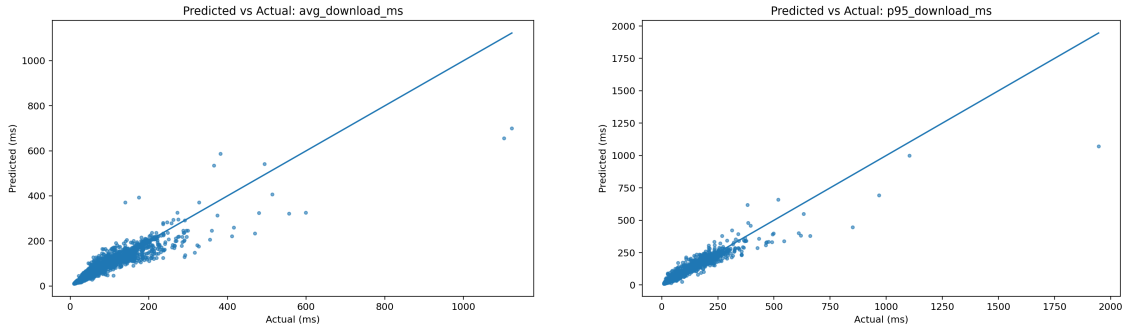


Figure 6.10: Predicted vs actual download latency (Random Forest Regression)

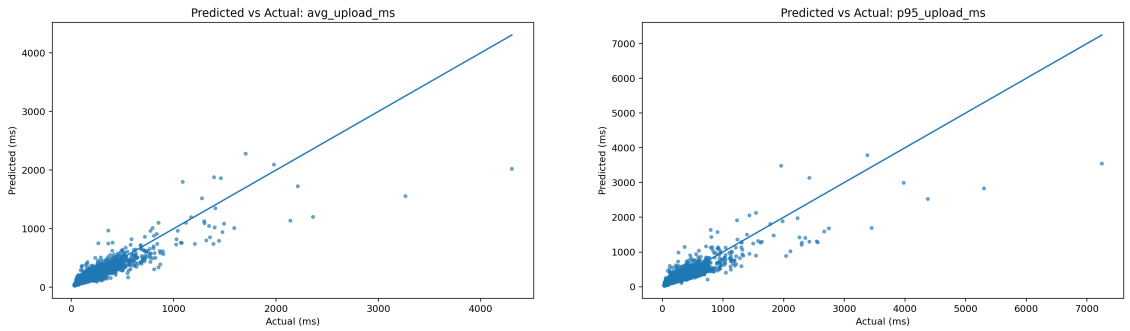


Figure 6.11: Predicted vs actual upload latency (Random Forest Regression)

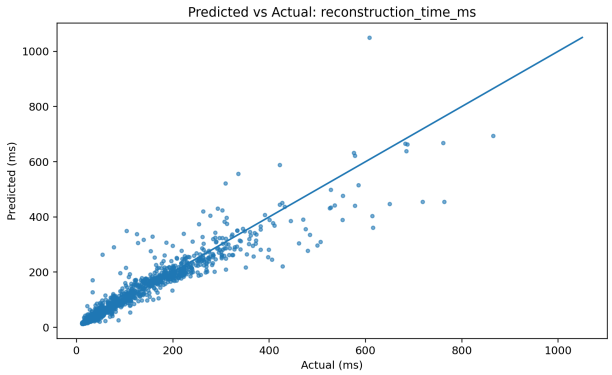


Figure 6.12: Predicted vs actual reconstruction time under failure (Random Forest Regression)

Here, the predicted vs actual points in the graph show a strong alignment. The closest agreement is seen in the reconstruction_time_ms, p95_download_ms, and avg_download_ms. These features are also consistent with their good $R^2$ scores (0.934, 0.931, 0.891). In the upload sector for features, good trend capture (avg_upload_ms $R^2$=0.813, p95_upload_ms $R^2$=0.783) is also found. This is as expected from heavy-tailed and sensitive storage latency distributions. In summary, this model can successfully capture emerging strong trends for policy making with a good sense of trade-offs for overall efficiency.

The regret histogram and regret CDF are given in Figure 6.13-

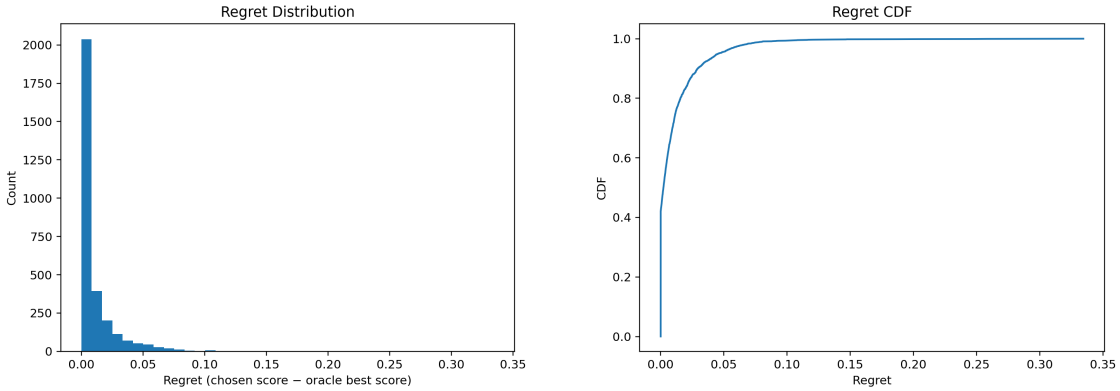


Figure 6.13: Regret histogram for upload latency prediction and Regret CDF for upload latency prediction

These regret distributions and CDF show that Random Forest–based policy recommendation system performs reliably for almost every workload situation. A strong concentration of regret values near zero in the histogram indicates a negligible cost added to the selection policy compared to oracle's best. CDF further confirms it with a steep rise at low regret values. But a long visible tail in both graphs mean in a small fraction of workloads have a noticeable increased regret, indicating the occasional mis-rankings under complex or extreme workload conditions.

For classifier-only design, we provided some graphs in Figure 6.14, 6.15, 6.16, 6.17, 6.18 and 6.19 to illustrate the comparisons between 3 classification models- CatBoostClassifier,

XGBoostClassifier and Logistic Regression.

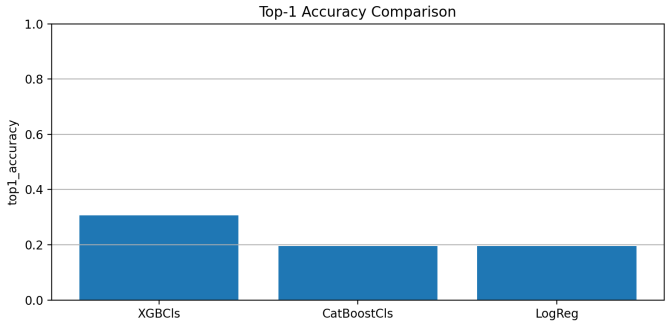


Figure 6.14: Top-1 accuracy comparison (CatBoostClassifier, XGBoostClassifier and Logistic Regression)

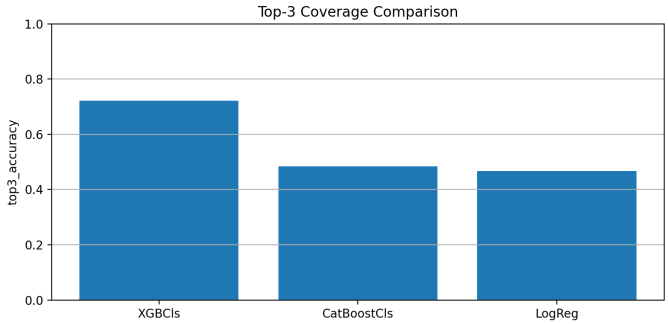


Figure 6.15: Top-3 accuracy comparison (CatBoostClassifier, XGBoostClassifier and Logistic Regression)

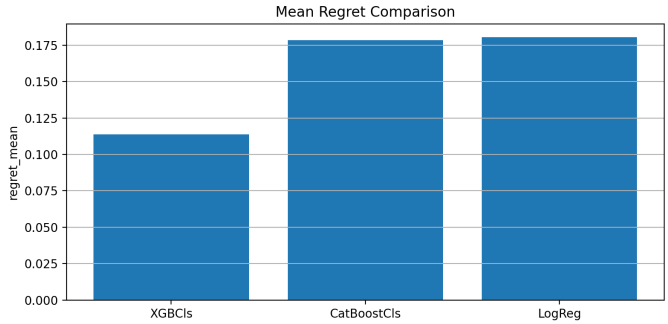


Figure 6.16: Mean regret comparison (CatBoostClassifier, XGBoostClassifier and Logistic Regression)

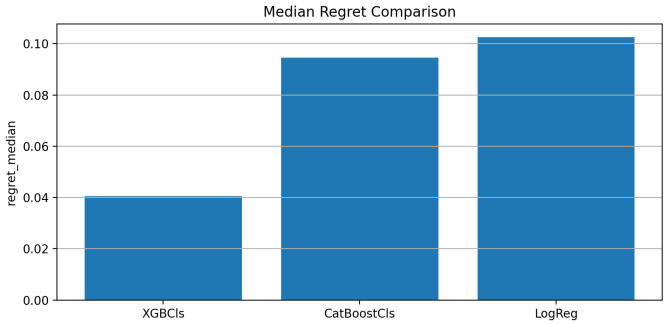


Figure 6.17: Median regret comparison (CatBoostClassifier, XGBoostClassifier and Logistic Regression)

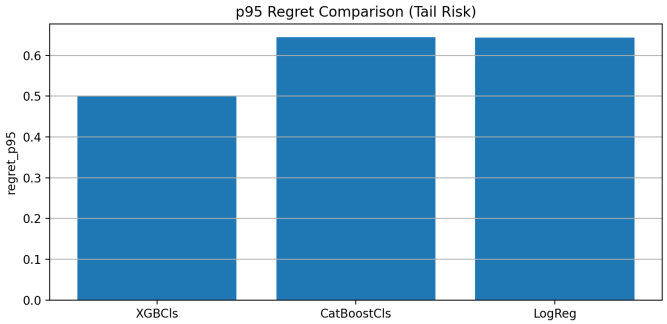


Figure 6.18: P95 regret comparison (CatBoostClassifier, XGBoostClassifier and Logistic Regression)

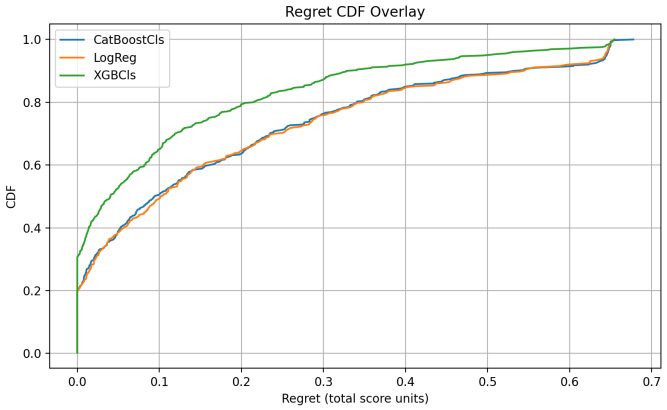


Figure 6.19: Regret CDF Overlay (CatBoostClassifier, XGBoostClassifier and Logistic Regression)

### 6.1.2 Quantitative metric

For regression-based models, Quantitative evaluation was performed using standard regression metrics. Here, we evaluated Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and coefficient of determination ($R^2$).

**Policy Recommendation Evaluation:**

For evaluating the policy recommendation system, the calculated metrics were:

- **Actual regret:** Difference of value between the score of the model-predicted best policy and the true best policy.
- **Top-$k$ accuracy:** The probability of the model predicting the optimal policy within the top-$k$ ranked policies.
- **actual_regret_mean:** Mean of all actual regret values.
- **actual_regret_median:** Median of all actual regret values.
- **p95_regret_median:** The performance loss observed in the worst 5% of cases.

**Regression Results**

For CatBoost Regressor, The experimentation results have been provided in the table below:

| **Target** | **MAE** | **RMSE** | $R^2$ |
|---|---|---|---|
| avg_upload_ms | 54.21 | 124.76 | 0.3978 |
| p95_upload_ms | 85.20 | 191.19 | 0.3682 |
| avg_download_ms | 21.69 | 53.02 | 0.1654 |
| p95_download_ms | 34.11 | 76.94 | 0.1658 |
| reconstruction_time_ms | 32.48 | 63.82 | 0.5425 |

Table 6.1: Regression performance metrics for predicted target variables (CatBoost)

Here, the upload and reconstruction metrics show moderate predictability scores and downloads have lower $R^2$ thus harder to fit and predict. As for policy recommendation quality, the results showcase -

- **Top-1 accuracy** = 47.17%
- **Top-3 accuracy** = 87.17%
- **actual_regret_mean** = 1.46
- **actual_regret_median** = 0.063
- **actual_regret_p95** = 10.23

These mean that for 47.17% of workloads, the model predicts the true best policy and in 87.17% of the cases it shows among the top-3 options. The median regret being 0.063 indicates that half of the workloads will always receive optimal rerecommendations. The worst 5% are shown as costly mistakes by the p95 regret. The point to be noted here is despite regression fit not being perfect, the decision quality

is high due to the policy selection depending more on relative ordering of candidates.

For XGBoost, similar results were observed. The following table shows the relevant metrics-

| **Target** | **MAE** | **RMSE** | $R^2$ |
|---|---|---|---|
| avg_upload_ms | 55.88 | 124.63 | 0.3991 |
| p95_upload_ms | 87.85 | 191.46 | 0.3666 |
| avg_download_ms | 22.65 | 52.92 | 0.1688 |
| p95_download_ms | 35.26 | 77.10 | 0.1626 |
| reconstruction_time_ms | 33.17 | 63.31 | 0.54997 |

Table 6.2: Regression performance metrics for predicted target variables (XGBoost)

From this table, we can see that reconstruction has the best predicted outcome, as $R^2$ has a moderate value of 0.54. This is likely because reconstruction time is strongly driven by a clear signal. For both upload latency both results are moderate. For both download latencies, the result is bad, and this indicates they are the hardest to predict. In every target feature, RMSE values are higher than MAE confirms the presence of heavy-tailed outliers.

- **Top-1 accuracy:** 48.16%
- **Top-3 accuracy:** 79.66%
- **actual_regret_mean:** $\approx 1.806$
- **actual_regret_median:** 0.089
- **actual_regret_p95:** $\approx 11.557$

These results indicate that the recommendation system selects optimal EC policy 48.16% of the time and optimal policy suggestions with in top-3 is 79.66% of workloads. The median regret (0.089) shows that the performance loss from not picking the true best policy is very small, and the mean regret (1.806) shows that the average penalty is overall low. However, the p95 regret ($\sim$11.557) shows that a very small number of workloads are experiencing higher loss mainly because of tail latency or failure effects.

And for Random Forest Regression, the results are as follows-

| Target | MAE | RMSE | $R^2$ |
|---|---|---|---|
| avg_upload_ms | 29.71 | 68.56 | 0.813 |
| p95_upload_ms | 50.41 | 116.02 | 0.783 |
| avg_download_ms | 5.58 | 17.63 | 0.891 |
| p95_download_ms | 5.98 | 19.98 | 0.931 |
| reconstruction_time_ms | 4.48 | 18.58 | 0.934 |

Table 6.3: Regression performance metrics for predicted target variables (Random Forest)

For policy recommendation, Random Forest Regression gives -

- **Top-1 accuracy** = 41.97%
- **Top-3 accuracy** = 100%
- **actual_regret_mean** $\approx$ 0.010139
- **actual_regret_median** = 0.002167
- **actual_regret_p95** $\approx$ 0.045648

This means that Random Forest Regression in quantitative evaluation shows a high predictability on the 20% holdout set. The result is captured in reconstruction_time_ms (MAE=4.477 ms, RMSE=18.575 ms, $R^2$=0.934) and download tail latency (p95_download_ms, $R^2$=0.931). Upload metrics are stronger (avg_upload_ms $R^2$=0.813, p95_upload_ms $R^2$=0.783), meaning higher variability. RMSE larger than MAE means occasional outliers consistent with heavy-tailed latency in object storage systems. Random Forest only chooses the oracle best policy only 42% of the time (Top-1 accuracy = 41.97% ), but the top 3 are always in the best (Top-3 accuracy = 100%) as even when the top choice is not optimal, the weighted penalties are rather small (actual_regret_mean $\approx$ 0.010139, actual_regret_median = 0.002167). But the only penalty increase is seen only in tails cases (actual_regret_p95 $\approx$ 0.045648). In short, the Top-3 is non-discriminative, and Top-1 plus regret provides meaningful evidence of recommendation quality.

### Classification Results

For classifier-only design, below are the overall benchmark of CatBoostClassifier, XGBoostClassifier and Logistic Regression using top-1 accuracy, mean regret, median regret, p95 regret.

| Classifier | Accuracy | | Mean | Median | P95 |
|---|---|---|---|---|---|
| | Top-1 | Top-3 | Regret | Regret | Regret |
| CatBoostCls | 19.56% | 48.44% | 0.1786 | 0.0946 | 0.6452 |
| Logistic Regression | 19.56% | 46.67% | 0.1807 | 0.1026 | 0.6432 |
| XGBCls | 30.66% | 72.22% | 0.1137 | 0.0404 | 0.6432 |

Table 6.4: Classification model performance (Top-1 accuracy, top-3 accuracy, mean regret, median regret, p95 regret)

From the table 6.4, we can see that all 3 models show poor performance across all metrics. Here we can see that XGBoostClassifier achieved the highest top-k accuracy, for which top-1 is 30.66% and top-3 is 72.22%. XGBoostClassifier also achieved the lowest mean regret, median regret and p95 regret, which shows XGBoostClassifier performs overall better comparing to CatBoostClassifier and Logistic Regression.

### 6.1.3 Challenges and Limitations

The limitations of the regression models include the following-

- Because of the heavy-tailed nature of the dataset and the existence of extreme outliers, exact regression is a difficult task.
- The regret histogram and CDF show that a small number of workloads are showing predictions that show values different from the true best policy by a large margin, which enforces the need for top-3 instead of using only top-1.
- Since the regression models do not need to predict exact values, the pipeline performs well even though the $R^2$ is moderate, not perfect.

In case of classifier-only design, there are several reasons why all 3 models show poor results:

- Latency and reconstruction measurements can vary across different rows. Therefore, two policies may have identical scores for the same context. Small amount of noise can change the 'best policy' label in such situations.
- Oracle selection favors certain policies while some policies rarely appears. This affects the model's ability to learn minority-class patterns.
- Although reliability scaling helps to reduce noise, tail latencies (p95) remain difficult to use as stable learning signals, especially for low object counts.

## 6.2 Analysis of Design Solutions

In the case of the regression-based paradigm, the system enforces multi-model regression with weighted scoring, which produces predicted metrics and ranks them. It also supports various scenario-based experimentation, such as changing weights

with latency vs resilience. Also, policies can be added/removed and scored, and a trend can be studied. This makes it more flexible than a direct policy-predicting classifier. On the other hand, the classifier-only design is simple to deploy as the system returns a single recommended policy without requiring any additional optimization step at runtime. However, the classifier-only design is inherently less flexible as the "best policy" depends on the chosen scoring definition. Also, the classifier learns from oracle labels that can be unstable when multiple policies have very similar scores under a workload context. This results in low top-1 accuracy.

## 6.3 Final Design Adjustments

For enhancing the training process and results, policy evaluation was switched to real-row grouped evaluation to fix any mathematical inconsistencies. Also, log1p transformation was used for targets and leaky features were avoided. Reconstruction was also confined within only failure rows for original values.

## 6.4 Statistical Analysis

### 6.4.1 Distribution of Prediction Errors

For CatBoost, the $R^2$ values that are 0.16-0.40 for latency metrics and 0.54 indicate that the prediction accuracy is imperfect, but the error magnitude must be judged in the context of storage system performance metrics which have inherent variability and a heavy-tailed nature. Due to resource contention, background queuing and recovery processes, latency shows a high variance. NMAE and NRMSE shows that the values remain within a bounded range and show stable behaviour.

For XGBoost we got similar results as CatBoost with the results show that prediction accuracy varies across target featuers, with $R^2$ values from 0.162 to 3.99 for latency metrics. Upload metrics achieved moderate variance while download metrics are harder to explain. Reconstruction time achieved highest predictability.
In the case of RandomForest Regression, prediction error shows stronger fit with $R^2$ values range from 0.783 to 0.934. This indicates higher variance in latency. Low errors are observed due to the small gap between MAE and RMSE. This is consistent behaviour from heavy tailed object storage latency. Overall, high $R^2$ means stable download and reconstruction with learnable structure in the data.

As for the classification paradigm, top-1 accuracy shows that the model does not select the best policy in most cases. This suggests that the policy classes are not strongly correlated using the available context features.

### 6.4.2 Regret Distribution Analysis

For CatBoost, median regret of 0.063 confirms that atleast in half of the cases, the workload variations would have the optimal policy recommended to them. The bounded mean regret showcases that the performance loss is limited even if the optimal policy is not chosen. Regret histogram and CDF confirm that there are rarely

any large deviations and very few samples have high regret values thus demonstrating that the recommendation system avoids suboptimal choices consistently.

Moving on to XGBoost, the evaluation shows that the median regret is 0.089 and the mean regret is 1.806, which demonstrates that performance loss happens when the exact best policy is not selected. The 95th percentile regret of 11.557 shows that the large regret events happen only in a small fraction of workload contexts.

As for Random Forest Regression, regret-based models show the true cost when recommended policy varies from oracle best policy for the same workload context. The histogram and the CDF of the frameworks are the evidence that the mistakes are frequently occurring and showing how often recommendations are close to optimal. Due to the strong fit of the regression model across various targets the ranked costs remained stable supporting low regret in most workloads.

In case of classification, median regret indicates that the typical penalty is often moderate when the model's recommendation is suboptimal. Mean regret is substantially higher than the median, which means that the regret distribution is right-skewed and a small number of contexts exist where the model makes significantly worse recommendations, which increases the average. P95 regret values show that in the worst 5% of cases, the chosen policy can incur a high additional cost relative to the oracle best. Overall, these results show that a classifier-only design provides limited reliability as an automated policy selection mechanism, especially when evaluated strictly against oracle best-policy labels.

### 6.4.3 Top-K Recommendation Reliability

For regression-only paradigm, CatBoost gives a top-1 accuracy of 47.17% and top-3 accuracy of 87.17%, which shows that statistically the model can accurately capture and determine the relative ordering of the policies even if the exact ranking is ambiguous.

Similarly, XGBoost achieves almost the same results as CatBoost, with 600 test context reports a top-1 accuracy of 48.16% and a top-3 accuracy of 79.66%, which shows that statistically the model can accurately capture and determine the relative ordering of the policies even if the exact ranking is ambiguous.

Random Forest Regression achieves a top-1 accuracy of 44.50% and a top-3 accuracy of 100%, indicating that although exact best-policy identification may vary, the model consistently includes the optimal policy within shortlist recommendations. This reflects strong ranking stability due to ensemble averaging across workload and system performance variations.

For classification-only paradigm, both CatBoostClassifier and Logistic Regression achieves a top-1 accuracy of 19.56%, while XGBoostClassifier achieves the highest top-1 accuracy of 30.66%. For top-3 accuracy, CatBoostClassifier and Logistic Regression respectively achieve 48.44% and 46.67%, while XGBoostClassifier achieved the highest top-3 accuracy of 72.22%. This shows that classification models are

failing to suggest the best policy most of the time.

## 6.5 Interpretation of Statistical Findings

The result shows that policy recommendation quality is better explained by top-k accuracy and regret metrics. Because of latency and tail-latency metrics that are heavy-tailed, models show imperfect prediction accuracy while still keeping the relative performance trends across policies. In our framework, the final ranking is dependent on a normalised weighted score. For the regression-based pipeline, we see that top-3 accuracy is high, indicating that the framework can rely on a shortlist of near-optimal policies, even though top-1 accuracy is not perfect. The median regret values show that most workload context is either optimal or close to near-optimal. Also, this indicates that the typical penalty for not choosing the optimal is very small. Again, p95 regret shows that 5% of workload contexts can still produce noticeably worse recommendations, which is important for policy ranking. Finally, comparing both of the paradigms, the classification-only approach shows lower top-1 and top-3 accuracy. This means that oracle labels become unstable when policies have similar scores under noisy measurements. Therefore, the final key insights of evaluation are that regression based model followed by score-based ranking has better performance in EC policy recommendation than a direct EC policy classification-based approach.

## 6.6 Discussion

From our experiment, we can see the behavior of different modeling approaches by observing the results. It gives us an idea about how machine learning can relate to the characteristics of erasure-coded object storage systems. The discussion focusing on the effectiveness of regression-based performance modeling, and classification-based performance modeling can help to find or select the optimal erasure coding policy. The regression-based models demonstrated moderate prediction accuracy for latency and reconstruction metrics, as reflected by MAE, RMSE, and $R^2$ values. The prediction error was significant because of heavy-tail latency and multiple low-level system factors. In observation, in a classification-based approach, accuracy is lower than regression based approach and higher instability. This indicates that direct policy labeling is less effective for capturing continuous and workload-dependent performance behavior. The results indicate that the classifier-only implementation is not yet reliable enough to be treated as a fully autonomous policy selection mechanism. The reasons are label instability, class imbalance, and an incomplete feature view of OpenStack Swift's internal state. Therefore, while the framework is practical and deployable, the results suggest that further refinement is needed before it can be considered dependable in all operating conditions.

# Chapter 7

# Conclusion

## 7.1 Summary of Findings

This thesis explores the possibility of automation in policy selection in erasure coding (EC) for OpenStack Swift by using a ML based framework. The core target is to overcome the limitations of static and heuristic-based EC configurations that fail to account for various workload and user usage contexts for policy choice.

To investigate this problem, a dataset with 30,000 operation records with 10 different EC policies with 3,000 distinct is created. The dataset covers various system behaviour, real client-visible latency, tail latency, and other metrics for successful and failure scenarios in the Swift All-In-One (SAIO) testbed.

Two distinct approaches are taken on this ML framework for selection on regression-based and classifier-based policy selection approach.
The regression-based approach works best with strong effectiveness, but with difficulty in precise latency prediction. The models are able to preserve policy ordering with Catboost and XGboost, achieving moderate regression accuracy against Random Forest Regression, which achieves substantially stronger predictive performance across all target metrics.

Despite imperfection in regression fitting, the recommendation quality remains high with top-1 accuracy remaining bet 42%-56% and top-3 always exceeding 79% accuracy. Regression-based results further prove that near-zero median regrets, even if the chosen policy is not perfect, it will be very close to the optimal one. Regret histograms and cumulative distribution functions confirmed that only in a tiny fraction of the workload large performance losses occur.

In contrast, the classifier-only approach's accuracy and reliability is limited, even though it is simpler to deploy. The result proved that classification for policy choice induces instability when multiple policies exhibit similar performance with class imbalance, and sensitivity to minor measurement noise.

In conclusion, the experiment proves that policy recommendations based on predicted performance metrics are significantly more robust and reliable than direct policy classification, especially when employed in complex storage systems where

performance differences between policies are often marginal and noisy.

## 7.2 Contributions to the Field

This research makes clear impactful contributions in the field of cloud object storage systems and intelligent storage management:

1. **Creation of a large-scale empirical EC performance dataset:**
   A novel structured dataset is created using real OpenStack Swift executions to capture real workload executions, system state, failure conditions, and multi-dimensional performance metrics. Unlike traditional EC studies, this research tries to directly link workload and user context to erasure coding policy selection.

2. **Demonstration of ML-based EC performance modeling in real systems:**
   This work tries to prove that ML frameworks can successfully learn relationships, trends and patterns in erasure-coded storage systems even with heavy-tailed latency distributions, data corruption, excessive noise and complex internal interactions.

3. **Comprehensive evaluation using regret-based metrics:**
   Unlike traditional studies, this research tried to focus on regret analysis, Top-k accuracy, and cumulative distribution evaluation to make a realistic, efficient decision where trade-offs have to be made.

4. **Comparative analysis of regression vs classification paradigms:**
   This thesis tries to give concrete evidence that classifier-only policy selection is fundamentally less stable in EC systems due to label noise and small inter-policy performance gaps, which sets the road for future automation systems.

5. **Foundation for adaptive storage control-plane automation:**
   By decoupling policy choice from static configurations and modelling system behavior the paper tries to establish a basis for future intelligent control-plane mechanisms in object storage platforms.

## 7.3 Recommendations for Future Work

Though the proposed framework proved strong feasibility, several approaches remain open for future research and enhancement:

1. **Online and adaptive learning mechanisms:** The research only reviews offline model training, online training remains unexplored in this research.

2. **Integration with Swift's control and monitoring layers:** Closer integration with Swift's internal telemetry will provide a richer UI experience with prediction stability and policy discrimination for the user.

3. **Evaluation on multi-node production-scale clusters:** Extending the framework to include multi-node environments will further allow the model validation under realistic network contention and in distributed recovery scenarios.

4. **Cross-platform generalization:** This framework can be further worked upon to be integrated into systems like Ceph or MinIO to evaluate portability across storage architectures.

# Bibliography


[1] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object storage: The future building block for storage systems," in *2005 IEEE International Symposium on Mass Storage Systems and Technology*, 2005, pp. 119–123. DOI: 10.1109/LGDI.2005.1612479.

[2] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[3] C. Huang et al., "Erasure coding in windows azure storage," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12, Boston, MA: USENIX Association, 2012, p. 2.

[4] en, Dec. 2013. [Online]. Available: https://ceph.io/en/news/blog/2013/benchmarking-ceph-erasure-code-plugins/.

[5] J. C. W. Chan, Q. Ding, P. P. C. Lee, and H. H. W. Chan, "Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, ser. FAST'14, Santa Clara, CA: USENIX Association, 2014, pp. 163–176, ISBN: 9781931971089.

[6] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," ser. CCS '15, Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 1322–1333, ISBN: 9781450338325. DOI: 10.1145/2810103.2813677. [Online]. Available: https://doi.org/10.1145/2810103.2813677.

[7] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794, ISBN: 9781450342322. DOI: 10.1145/2939672.2939785. [Online]. Available: https://doi.org/10.1145/2939672.2939785.

[8] X. Pei, Y. Wang, X. Ma, and F. Xu, "Efficient in-place update with grouped and pipelined data transmission in erasure-coded storage systems," *Future Generation Computer Systems*, vol. 69, pp. 24–40, Dec. 2016. DOI: 10.1016/j.future.2016.10.016. [Online]. Available: https://doi.org/10.1016/j.future.2016.10.016.

[9] J. Matt, P. Waibel, and S. Schulte, "Cost- and latency-efficient redundant data storage in the cloud," in *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*, 2017, pp. 164–172. DOI: 10.1109/SOCA.2017.30.

[10] Y. Su, D. Feng, Y. Hua, and Z. Shi, "Predicting response latency percentiles for cloud object storage systems," in *2017 46th International Conference on Parallel Processing (ICPP)*, 2017, pp. 241–250. DOI: 10.1109/ICPP.2017.33.

[11] V. Bucur, C. Dehelean, and L. Miclea, "Object storage in the cloud and multi-cloud: State of the art and the research challenges," in *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, 2018, pp. 1–6. DOI: 10.1109/AQTR.2018.8402762.

[12] F. Xu, Y. Wang, and X. Ma, "Incremental encoding for erasure-coded cross-datacenters cloud storage," *Future Generation Computer Systems*, vol. 87, pp. 527–537, Apr. 2018. DOI: 10.1016/j.future.2018.04.047. [Online]. Available: https://doi.org/10.1016/j.future.2018.04.047.

[13] R. R. Noel, R. Mehra, and P. Lama, "Towards self-managing cloud storage with reinforcement learning," en, *2019 IEEE International Conference on Cloud Engineering*, pp. 34–44, Jun. 2019. DOI: 10.1109/ic2e.2019.000-9. [Online]. Available: https://doi.org/10.1109/ic2e.2019.000-9.

[14] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, *Catboost: Unbiased boosting with categorical features*, 2019. arXiv: 1706.09516 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1706.09516.

[15] A. Chugh. "Mae, mse, rmse, coefficient of determination, adjusted r squared — which metric is better?" Accessed: Jan. 25, 2026. [Online]. Available: https://medium.com/analytics-vidhya/mae-mse-rmse-coefficient-of-determination-adjusted-r-squared-which-metric-is-better-cd0326a5697e.

[16] C. R. Harris et al., "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[17] D. R. Patil, "Dynamic resource allocation and memory management using machine learning for cloud environments," *International Journal of Advanced Trends in Computer Science and Engineering*, vol. 9, no. 4, pp. 5921–5927, Aug. 2020. DOI: 10.30534/ijatcse/2020/255942020. [Online]. Available: http://doi.org/10.30534/ijatcse/2020/255942020.

[18] T. pandas development team, *Pandas-dev/pandas: Pandas*, version latest, Feb. 2020. DOI: 10.5281/zenodo.3509134. [Online]. Available: https://doi.org/10.5281/zenodo.3509134.

[19] J.-J. Kim, "Erasure-coding-based storage and recovery for distributed exascale storage systems," *Applied Sciences*, vol. 11, no. 8, 2021, ISSN: 2076-3417. DOI: 10.3390/app11083298. [Online]. Available: https://www.mdpi.com/2076-3417/11/8/3298.

[20] A. S. Mondal, A. Mukhopadhyay, and S. Chattopadhyay, "Machine learning-driven automatic storage space recommendation for object-based cloud storage system," *Complex Intelligent Systems*, vol. 8, no. 1, pp. 489–505, Sep. 2021. DOI: 10.1007/s40747-021-00517-4. [Online]. Available: https://doi.org/10.1007/s40747-021-00517-4.

[21] M. Saadoon, S. H. A. Hamid, H. Sofian, H. H. Altarturi, Z. H. Azizul, and N. Nasuha, "Fault tolerance in big data storage and processing systems: A review on challenges and solutions," *Ain Shams Engineering Journal*, vol. 13, no. 2, p. 101 538, Jul. 2021. DOI: 10.1016/j.asej.2021.06.024. [Online]. Available: https://doi.org/10.1016/j.asej.2021.06.024.

[22] Z. Su et al., "Secure and efficient federated learning for smart grid with edge-cloud collaboration," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 2, pp. 1333–1344, Jul. 2021. DOI: 10.1109/tii.2021.3095506. [Online]. Available: https://doi.org/10.1109/tii.2021.3095506.

[23] G. Levitin, L. Xing, and Y. Dai, "Optimizing uploading and downloading pace distribution in system with two non-identical storage units," *Reliability Engineering System Safety*, vol. 231, p. 109 017, Dec. 2022. DOI: 10.1016/j.ress.2022.109017. [Online]. Available: https://doi.org/10.1016/j.ress.2022.109017.

[24] M. Akbar, I. Ahmad, M. Mirza, M. Ali, and P. Barmavatu, "Enhanced authentication for de-duplication of big data on cloud storage system using machine learning approach," *Cluster Computing*, vol. 27, no. 3, pp. 3683–3702, Nov. 2023. DOI: 10.1007/s10586-023-04171-y. [Online]. Available: https://doi.org/10.1007/s10586-023-04171-y.

[25] D. Durner, V. Leis, and T. Neumann, "Exploiting cloud object storage for high-performance analytics," *Proceedings of the VLDB Endowment*, vol. 16, no. 11, pp. 2769–2782, Jul. 2023. DOI: 10.14778/3611479.3611486. [Online]. Available: https://doi.org/10.14778/3611479.3611486.

[26] M. Liu, L. Pan, and S. Liu, "Cost optimization for cloud storage from user perspectives: Recent advances, taxonomy, and survey," *ACM Comput. Surv.*, vol. 55, no. 13s, Jul. 2023, ISSN: 0360-0300. DOI: 10.1145/3582883. [Online]. Available: https://doi.org/10.1145/3582883.

[27] "Managing machine learning lifecycle in oracle cloud infrastructure for erp-related use cases," *International Journal of Emerging Research in Engineering and Technology*, vol. 4, 2023. DOI: 10.63282/3050-922X.IJERET-V4I3P110. [Online]. Available: https://doi.org/10.63282/3050-922x.ijeret-v4i3p110.

[28] K. Mukherjee et al., "Towards optimizing storage costs on the cloud," *arXiv (Cornell University)*, Jan. 2023. DOI: 10.48550/arxiv.2305.14818. [Online]. Available: https://arxiv.org/abs/2305.14818.

[29] en, 2024. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2024-11-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-total-723-billion-dollars-in-2025.

[30] Y. Chen, Q. Ke, H. Li, Y. Wu, and Y. Zhang, "X meta: Ssd-hdd-hybrid optimization for metadata maintenance of cloud-scale object storage," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 2, pp. 1–20, Mar. 2024. DOI: 10.1145/3652606. [Online]. Available: https://doi.org/10.1145/3652606.

[31] J. Hu, J. Kosaian, and K. V. Rashmi, "Rethinking erasure-coding libraries in the age of optimized machine learning," in *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '24, Santa Clara, CA, USA: Association for Computing Machinery, 2024, pp. 23–30, ISBN: 9798400706301. DOI: 10.1145/3655038.3665943. [Online]. Available: https://doi.org/10.1145/3655038.3665943.

[32] E. A. T. Kamble, "Predictive resource allocation strategies for cloud computing environments using machine learning," *Deleted Journal*, vol. 19, no. 2, pp. 68–77, Jan. 2024. DOI: 10.52783/jes.692. [Online]. Available: https://doi.org/10.52783/jes.692.

[33] J. Pongthao, A. Na-udom, and J. Rungrattanaubol, "Machine learning classification with logistic regression feature selection approach on health datasets," *IAENG International Journal of Applied Mathematics*, vol. 55, no. 6, p. 25, 2024.

[34] H. A. Salman, A. Kalakech, and A. Steiti, "Random forest algorithm overview," *Babylonian Journal of Machine Learning*, vol. 2024, pp. 69–79, 2024, ISSN: 3006-5429. DOI: 10.58496/BJML/2024/007.

[35] S. Syed and E. M. Albalawi, "Optimizing cloud resource allocation with machine learning: A comprehensive approach to efficiency and performance," *Research Square (Research Square)*, Aug. 2024. DOI: 10.21203/rs.3.rs-4825637/v1. [Online]. Available: https://doi.org/10.21203/rs.3.rs-4825637/v1.

[36] Y. Wang, Q. Bao, J. Wang, G. Su, and X. Xu, "Cloud computing for large-scale resource computation and storage in machine learning," *Journal of Theory and Practice of Engineering Science*, vol. 4, no. 03, pp. 163–171, Mar. 2024. DOI: 10.53469/jtpes.2024.04(03).14. [Online]. Available: https://doi.org/10.53469/jtpes.2024.04(03).14.

[37] M. Cui and Y. Wang, "Ec-kad: An efficient data redundancy scheme for cloud storage," *Electronics*, vol. 14, no. 9, 2025, ISSN: 2079-9292. DOI: 10.3390/electronics14091700. [Online]. Available: https://www.mdpi.com/2079-9292/14/9/1700.

[38] Griffiths, *The latest cloud computing statistics (updated january 2025) — aag it support*, en-GB, Jan. 2025. [Online]. Available: https://aag-it.com/the-latest-cloud-computing-statistics/.

[39] J. Noor, R. I. Upoma, M. S. I. Sakif, and A. A. A. Islam, "Towards benchmarking erasure coding schemes in object storage system: A systematic review," *Future Generation Computer Systems*, vol. 163, p. 107 522, 2025, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2024.107522. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X24004862.

[40] Z. Shen et al., "A survey of the past, present, and future of erasure coding for storage systems," *ACM Trans. Storage*, vol. 21, no. 1, Jan. 2025, ISSN: 1553-3077. DOI: 10.1145/3708994. [Online]. Available: https://doi.org/10.1145/3708994.

[41] en. [Online]. Available: https://docs.openstack.org/swift/latest/overview_policies.html.

[42] en. [Online]. Available: https://docs.openstack.org/swift/latest/overview_erasure_code.html.

[43] "Train series release notes — swift release notes documentation," Accessed: Jan. 29, 2026. [Online]. Available: https://docs.openstack.org/releasenotes/swift/train.html.