Benchmarking Erasure Coding Schemes In OpenStack Swift

by

Khandaker Ishrak Noor
19101060
Yeasif Bin Noor
19101085
Shaima Afrin
19101641
Mahanaj Hossain
18201017
Rezuana Imtiaz Upoma
19101130

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

Bachelor of Science in Computer Science and Engineering
Department of Computer Science and Engineering
School of Data and Sciences
Brac University
May 2023

Declaration

It is hereby declared that

1. The thesis submitted is 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:

Khandaker Ishrak Noor

Khandaker Ishrak Noor

19101060

Yeasif Bin Noor

Yeasif Bin Noor

19101085

Shaima Afrin

Shaima Afrin

19101641

Mahanaj Hossain Uday

Mahanaj Hossain

18201017

Rezuana Upoma

Rezuana Imtiaz Upoma

19101130

Approval

The thesis titled "Benchmarking Erasure Coding Schemes In OpenStack Swift" submitted by

1. Khandaker Ishrak Noor (19101060)
2. Yeasif Bin Noor (19101085)
3. Shaima Afrin (19101641)
4. Mahanaj Hossain (18201017)
5. Rezuana Imtiaz Upoma (19101130)

Of Summer, 2022 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science and Engineering on May, 2023.

Examining Committee:

Supervisor:
(Member)

[illegible]

Jannatun Noor
Senior Lecturer
Department of Computer Science and Engineering
School of Data and Sciences
Brac University

Thesis Coordinator:
(Member)

Md. Golam Rabiul Alam, PhD
Professor
Department of Computer Science and Engineering
School of Data and Sciences
Brac University

Head of Department:
(Chair)

Sadia Hamid Kazi, PhD
Chairperson and Associate Professor
Department of Computer Science and Engineering
School of Data and Sciences
Brac University

Abstract

Erasure coding (EC) is a security measure that allows for data to be reconstructed from parity pieces, which eliminates the need for complete data replication. EC offers increased data redundancy, efficiency, lowers storage cost and boosts fault tolerance, making it preferable to replication in Swift. The basic idea is to encrypt a certain amount of data in a way that guarantees that all coded pieces are transferred without any loss. The time efficiency of EC methods becomes increasingly important in guaranteeing optimal system performance as data volumes continue to increase rapidly. A number of variables, such as the particular algorithm used, data size, the number of storage nodes, hardware resources, and network conditions, can affect how quickly EC works. The primary subject of our analysis was erasure coding algorithm- Reed-Solomon Codes. The study investigates the encoding speed of the algorithm, considering factors like data size and the number of parity blocks generated. In the context of addressing time efficiency and fault tolerance challenges in cloud-based object storage systems, our paper focuses on evaluating and improving existing mechanisms. It comprehensively analyzes time efficiency mechanisms, such as data placement policies, and scheduling algorithms, to enhance data retrieval and storage processes. Exploring the time efficiency of EC is also focused where it is conducted as an analysis of the time it takes for a cloud storage system to store data by examining two datasets and determining the duration it takes to store those same dataset files on the cloud storage system (Swift). It also assesses fault tolerance mechanisms, including redundancy schemes, error correction codes and distributed data placement strategies to improve system resilience. The research proposes innovative approaches to minimize access latency, improve overall time efficiency and ensure data availability even in the presence of failures.

Keywords: Erasure coding; Swift; Reed-Solomon Codes; Cloud Storage System; Time Efficiency; Fragments

Dedication

We devote this report to our respected faculties, whose direction, instruction and assistance made it possible for us to complete it. This would not have been possible without their help and inspiration and we are really grateful to them.

Acknowledgement

First and foremost, all praises go to the Almighty Allah, to whom we owe the completion of our thesis without substantial disruption.

After that, we would like to express our gratitude to our supervisor, Jannatun Noor, for her advice and unwavering support throughout this entire process. Additionally, we want to thank Md. Sadiqul Islam Sakif, a machine learning engineer at mPower Social enterprise. We also would like to thank Joyanta Jyoti Mondal, from University of Alabama at Birmingham, for their invaluable assistance and unwavering support.

At the end, we would want to express our respect to our parents for their continuous support and prayers, which have enabled us to reach this point in our pursuit of graduating.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing has undergone a paradigm shift in recent years due to the difficulty and importance of storing large amounts of data in decentralized storage networks. OpenStack Swift [11], one of the platforms, is working on such a security system by following an algorithm that will aid the user in creating a system that would adequately protect and provide access to all data. The bulk of data is growing exponentially as new technologies are introduced, making it more big data. The execution layer in a typical big-data system is responsible for coordinating the various stages of job completion.

It controls the runtime environment, the applications that users may use to alter, analyze data and the distributed storage that stores and provides access to massive volumes of data. Therefore, the big data systems' underlying framework is the distributed storage layer [23]. What we mean by "Big Data" is a massive amount of data that is continually expanding. As a result of its complexity and size, no existing specialist data management systems are up to the task of storing or processing this data. OpenStack Swift is a free and open-source cloud computing platform designed to handle massive volumes of data. Erasure Coding is the name of the algorithm used to build the security mechanism.

Erasure coding [3] is used to secure data in distributed storage since it is both trustworthy and efficient. Separating data files into data and parity blocks and encrypting them makes it possible to retrieve the primary data even if part of the encoded data is lost. Data security in horizontally scalable distributed storage systems is achieved by the use of erasure coding, which involves storing encoded data across a large number of disks and nodes. When data corruption or the failure of a node occurs, the data may be restored from blocks stored on other nodes. Considering that it erases information across nodes and drives, erasure coding is a considerable improvement over previous systems when it comes to dealing with the same number of drive failures. This technique may be implemented in OpenStack Swift to further optimize the process. Using erasure coding, we can create a more robust security infrastructure in OpenStack Swift. However, a more secure method of storing such massive amounts of data may provide its own set of difficulties. A high-security system requires data with consistent regulations. With Openstack's Erasure coding, several object rings must be constructed, which requires restricted degrees of segmentation of clusters, as mentioned by the storage regulations that are implemented. In addition, the rings will ultimately decide the most optimal means of data storage and where that data should be located. Big data of a certain database type is included in each of the rings. For such files to be safe on OpenStack, fast has to adhere to the storage rules set out by the cloud provider.

1.1 Motivation

In the modern, data-driven world, it is very necessary to have storage that is reliable as well as effective in order to be able to access data. Traditional methods have both a limited capacity and a low fault tolerance, both of which are problems that need to be solved. Erasure coding, which is more often referred to by its acronym EC, is a solution that offers time efficiency, user friendliness and fault tolerance. This makes it a desirable choice to consider. EC's technique of dispersing the data over numerous devices and producing redundant parts ensures that the data's integrity will be preserved even in the event that one or more of those devices fails. Because of its capacity for fault tolerance, the possibility of data being lost is reduced. In addition, EC's lower storage overhead enhances both the performance of the system as well as the user experience. This is due to the fact that it offers redundancy while maintaining a smaller footprint. Since of its transparent data recovery, it is easy to use since it removes the need for significant system modifications or user training. This makes it user-friendly. Since this helps companies to make intelligent judgments while also contributing to the development of data storage, it is of the highest significance to carry out an examination of the performance of erasure coding schemes. Because of the promise that it would do away with the need for redundant storage, erasure coding has emerged as an enticing solution for the challenges of fault tolerance and time efficiency.

1.2 Research Problem

The need for effective and dependable data storage and retrieval procedures grows as cloud storage systems develop and manage larger amounts of data. A potential method for fault-tolerant data storage in cloud systems has emerged which is erasure coding. It is still difficult to strike a balance between time economy and fault tolerance, however. This study seeks to thoroughly examine and improve the fault tolerance and time efficiency properties of erasure coding in the Swift cloud storage system. This work seeks to give insights and useful suggestions to improve the performance and reliability of erasure coding techniques in cloud storage systems by comprehending the influence of different elements and investigating novel ways.

Recognizing the Fault Tolerance and Time Efficiency Issues in Erasure Coding for Swift Cloud Storage Systems:

Erasure coding (EC) is a powerful technique utilized in cloud storage systems to ensure data stability and fault tolerance. By dispersing data across multiple storage nodes and creating redundant pieces, EC can withstand a certain number of node failures while enabling efficient data recovery [38]. In the vast and error-prone landscape of cloud storage, EC plays a critical role in guaranteeing data availability and longevity. Swift, an open-source object storage system designed for scalable and accessible cloud storage, relies heavily on erasure coding to ensure fault tolerance and effective data retrieval. However, the challenge lies in striking a balance between time efficiency and fault tolerance. The temporal efficiency of erasure coding refers to the computational cost involved in encoding and decoding processes, impacting the speed of storage and retrieval. Meanwhile, fault tolerance relates to the system's ability to recover data in the face of node failures or data corruption, influenced by the quantity of redundant pieces, redundancy strategy and recovery effectiveness [37]. To achieve effective and reliable data storage and retrieval in Swift cloud storage systems, it is crucial to evaluate existing erasure coding methods.

Time efficiency is crucial in erasure coding as it directly affects system performance and user satisfaction. Whether it's in cloud storage systems or distributed storage systems like HDFS and Ceph, erasure coding is used to minimize storage overhead while safeguarding against data loss. Data is fragmented into smaller pieces, such as blocks or objects to expedite retrieval and recovery. However, several factors can impede time efficiency, including the complexity of encoding and decoding, network and storage latency, hardware limitations, implementation inefficiencies, and sub-optimal data fragmentation [40]. By addressing these factors and optimizing the erasure coding process, faster data access and reconstruction can be achieved while minimizing delays and resource consumption.

This evaluation involves assessing time efficiency and fault tolerance properties of selected Swift erasure coding algorithms to identify strengths, weaknesses and areas for improvement. By carefully analyzing these algorithms, valuable insights can be gained, leading to further research and advancements in this domain. The subsequent sections of this article will cover the selection process for erasure coding methods, evaluation metrics, methodology, outcomes, discussion of findings and a comparison with relevant work. Through this comprehensive examination, we aim to contribute insights and recommendations to enhance the performance and dependability of Swift-based cloud storage systems.

1.3 Research Objectives

The primary focus of this paper is to address the critical challenges of time efficiency and fault tolerance in cloud-based object storage systems. To achieve this objective, the research will undertake a comprehensive analysis of existing time efficiency mechanisms in cloud-based object storage, aiming to identify their limitations and areas for improvement. By investigating factors such as data access patterns, storage architectures and network conditions, the research will propose innovative strategies and optimizations to enhance time efficiency. In addition to time efficiency, the paper will also focus on assessing the fault tolerance mechanisms currently employed in cloud-based object storage systems. This evaluation aims to determine the effectiveness of these mechanisms in mitigating failures and ensuring data availability.

The research will explore various fault tolerance techniques, such as data redundancy schemes, error correction codes, and distributed data placement strategies, to enhance the system's resilience to failures and improve fault tolerance. To evaluate and validate the proposed techniques, extensive experimental evaluations will be conducted. The research will involve comparative analysis with existing approaches to quantify the improvements achieved in terms of time efficiency and fault tolerance. Performance metrics, such as response time, throughput and recovery time will be measured and analyzed to provide quantitative evidence of the proposed enhancements.

1.4 Our Contributions

In this section, we discuss the contributions made in our thesis work. Our primary focus is on investigating the time efficiency of erasure coding in the context of Cloud-based Storage System Swift. We measure the upload, download and delete times for three specific erasure coding schemes namely Reed Solomon (5+3), Reed Solomon (10+4), and Reed Solomon (7+5) [1]. This analysis provides valuable insights into the time efficiency of different erasure coding schemes, which has not been extensively compared in previous research.

Additionally, we explore the fault tolerance aspect of erasure coding using a simulator called SimEDC. This simulator allows us to evaluate the fault tolerance of numerous erasure coding schemes, further enhancing the comprehensiveness of our study. We compare the fault tolerance results with various EC schemes, providing valuable findings for improving the reliability and fault tolerance of erasure coding.

Furthermore, our work stands out as we utilize a unique dataset created by us for the data analysis. The dataset, named MCSD-100 [39], ensures that our analysis is based on distinct and relevant data. Additionally, we incorporate an existing benchmark dataset, Coco-17 [4], for further comparisons and validation.

To validate the uniqueness of our work, we compare our results with other relevant papers in the field. Our comparative analysis concludes that our research is distinct and offers numerous possibilities for enhancing the time efficiency and fault tolerance of erasure coding.

In summary, our contributions include an in-depth analysis of time efficiency for different erasure coding schemes, exploration of fault tolerance using a simulator, utilization of a self-created dataset, comparison with other papers and identification of opportunities for improving erasure coding techniques. These contributions enhance the understanding and potential advancements in the field of erasure coding for Cloud-based Storage Systems.

1.5 Thesis Organization

Here, this portion provides a comprehensive description of every chapter of our thesis paper. After completing this chapter's introduction, problem statement and contributions section, we now list the other chapter of our thesis paper below:

- Chapter 2 conducts the Literature Review where we list relevant works that we have gathered from numerous articles that are related to our thesis work, as well as the Background Study which displays the technologies or architectures we utilized and read for our research.
- Chapter 3 covers the background of our selected work where we portray the information of our selected methods & simulator.
- Chapter 4 represents the methodologies of our work.
- Chapter 5 represents the dataset we used in our work.
- Chapter 6 represents the implementation of the methods ,simulator & the experimental result we analysed in used in our work.
- Chapter 7 concludes our thesis with future plan.

Chapter 2

Literature Review

In the field of Cloud Computing. It has become a trend and a new challenge in terms of developing an improved security system in a distributed storage system. Many challenges with new technologies are overcome one day after another. But behind all those successes, many researchers have studied and developed different theories and methodologies. Storing Big Data in OpenStack Swift with the help of Erasure Coding has also been successful in many occasions. Those achievements were pursued as previous authors have mentioned and came up with different ideas and techniques. Such Reviews are given below:

Wang et al. [23] analyzes and compares different Distributed Storage Systems (DSS) adopted by Amazon, Google and Microsoft Azure to deal with the celestial charge of data. By examining the shortcomings of replication and other design principles in the distributed storage system, the author attempts to demonstrate how erasure coding can be a good substitute for such principles. Examination of distributed storage systems has revealed a number of problems, including security, data placement and migration costs, and the expense of holding replicated data after implementing different principles on distributed storage system. Some of these problems can be overcome through erasure coding.

Emu at al. [20] offers a significant improvement to shared-mode resource allocation for cloud-based load testing that ensures the cloud's virtual machine resources may be used in an economical manner. As their cloud server, OpenStack Swift was first set up. In order to conduct the testing procedure, they have also created a few test scenarios. On the basis of metrics like load, protocol, file size, and URL, the test cases are organized. Then, they conduct the testing for our test cases using Apache JMeter as our load testing tool. TC2 had the fastest average reaction time of 4.9ms. The scenario used for the received test was a little file, the same URL, and no load. This was for TC5, which included a sizable file, a unique URL, and a load. Here, they have increased the amount of users to determine the servers' load capacitance sealing capacity. The major goal of their study is to use the testing tool to examine how the system would behave under various conditions and then compare the results.

Nandyal et al. [15] focuses on data availability and stability in distributed file systems, and a variety of strategies have been employed. The author mentions a method for storing data in a distributed file system based on the replication technique being used, but more recently, an erasure-coding (EC) technique has been applied due to the issue of space efficiency. The space efficiency problem is improved more by the EC method than by replication. However, the EC approach suffers from a number of performance degradation issues, including input and output (I/O) deterioration and encoding and decoding degradation. In order to perform multiple I/O requests that came in during encoding in an EC-based distributed file system, this study recommends a buffering and combining method. In order to distribute the disk input/output loads created during decoding, the report also suggests four recovery measures: random block layout, multi-thread-based parallel recovery, matrix recycle approach, and disk input/output load distribution.

Tanwar et al. [22] concentrates on Data partitioning and deployment are necessary for distributed data management which is performed at the data storage level. The distributed system's overhead is closely tied to the data partitioning approach. In this paper, the author proposes The distributed linear order partition (DLOP) based on timestamps, along with a widely dispersed storage and processing mechanism to choose the best data splitting and updating technique for the given application. Two different partitioning techniques are suggested in this scheme, both of which are based on the characteristics of an "equivalent division" of a linear order partition (LOP). One is partitioning based on time interval equilibrium. Another is partitioning based on query expectation. The author states that an index-based data query mechanism is consistently constructed at each site in the distributed system to complete the distributed administration of data. The study employs that, the relevant experiments confirm the viability and effectiveness of the suggested storage strategy and demonstrate that the suggested approach is effective for the data scale self scalability and lowers the cluster hardware configuration requirements.

Opara-Martins et al. [5] they have mentioned huge volumes of data generated by Social media, also referred to as big data. Big data storage requires a number of physical resources. To avoid the use of physical resources, cloud resources are being used. Amazon Web Services, OpenStack, Rackspace, and many other companies offer cloud resources. Big data that has been stored can be utilized for sentimental analysis, call center optimization, real-time fraud detection, and traffic management study. The authors state that there are currently no solutions for cloud-based big data analysis. To perform analyzing big data, users need to manually gather essential cloud storage resources and install necessary software which is difficult to perform in complex distributed services. To solve this issue, it is best to think of the services as a single application made up of virtual machines. Authors have proposed a solution to overcome this problem by implementing an openstack cloud system with a multi-node arrangement to offer Hadoop(an open source tool for processing and saving huge datasets quickly) as a service to the users. The OpenStack cloud environment's Hadoop service is easily accessible by users.

Zhang et al. [14] mainly focuses on Big Data Distributed storage systems that provide significant fault tolerance while requiring little storage overhead. Erasure-coded systems save space but require more operational computational complexity and network bandwidth. In this article, the author presents RAPID, a rapid data update technique that chooses a group of code blocks for modifications and adjusts the subset's robustness according to the expected number of failures. To guarantee consistency in data and code block updates, the solution uses a protocol that makes use of both locking and buffering methods.

Kengond et al. [11] emphasizes the difficulties presented by the massive amount of data produced by social media, often known as big data. Cloud resources like Amazon Web Services, OpenStack, and Rackspace are often utilized to store and analyze this data. The current big data analysis on the cloud solutions, however, need manual software installation and setup, which may be difficult and time-consuming. A multi-node OpenStack cloud system implementation has been created as a solution to this problem. With the help of this solution, users may simply access and make use of Hadoop in the OpenStack cloud environment.

Siagian et al. [17] establishes a comprehensive OpenStack cluster with diverse configurations and operational scenarios, accompanied by a thorough examination of its network capabilities and an elucidation of its underlying processes. It appears that OpenStack, also known as virtual machine installations utilize an online storage concentrate that requires authentication to store data storage files. The methodology employed explores different scenarios utilizing OpenStack virtual systems and cloud-based solutions to enhance network performance for a video surveillance database system architecture. This is achieved by utilizing the Sensor web Enabled structures and data storage in the cloud. Data can be acquired through various means such as training devices on camera/edge devices, record and metadata giving, and efficient cloud-based storage. The objective of this study is to evaluate the performance load of large-scale video surveillance in comparison to small ones. The aim is to demonstrate the effectiveness, scalability, and reliability of cloud-based data storage for smart MCS. One of the key features is that the storage is dynamically adjusted, which helps to ensure that there are no concerns about insufficient or wasted space. The paper offers an option for applying web applications for surveillance cameras video surveillance through an object cloud file storage system that is based on the widely-used open-source cloud operating system, OpenStack Swift. The system appears to be functioning well on OpenStack Swift in an Infrastructureas-a-Service environment. It effectively allocates resources for storing original data captured via various lens monitoring devices.

Heo et al. [6] covers the necessity for cloud storage systems built on solid-state drives (SSDs) to manage massive volumes of data coming from internet users and Internet of Things (IoT) devices. For high-performance and dependable storage, flash-based redundancy arrays of standalone disks (RAID) topologies are utilized. However, maintaining parity info with every write operation may cause SSDs to wear out more quickly. The article suggests a remedy dubbed "parity database reduction" for OpenStack cloud-based storage platforms using all-flash arrays to resolve this problem. This strategy concentrates primarily on eliminating superfluous parity data and putting it in the consistency discs of the all-flash array, in contrast to conventional deduplication techniques that completely eliminate duplicate data. By using this technique, the quantity of parity data written may be greatly reduced, reducing the load on SSDs and extending their useful life. Experimental findings validate the efficacy of the suggested strategy.

Lombardo et al. [30] presents a technique to tackle the problem of frequent updates of parity data in RAID storage based on flash memory, which may have an adverse effect on the longevity of solid-state drives. The proposed solution is referred to as "parity data deduplication." The paper centers on the utilization of all-flash arrays for OpenStack cloud block storage. The approach aims to selectively address redundant parity data by storing it in the parity storage devices of the all-flash array, rather than entirely eliminating duplicate data. The authors have presented a comprehensive analysis of their suggested approach, which includes experimental findings and an evaluation of its performance. The conducted experiments indicate that the proposed technique for parity data deduplication is efficient in reducing the frequency of parity data write operations when compared to conventional data deduplication methods. It is imperative to acknowledge that the document in question is an arXiv preprint along with could not have undergone a formal peer review process. Hence, it is recommended to thoroughly assess the content and discoveries as initial research.

Zhou et al. [24] proposes a methodology involving a data placement scheme utilizing top-down transmission. This is achieved through the integration of an enhanced metaheuristic algorithm, which effectively manages the pipelined data transmission across the network's tree structure. The proposed scheme has the potential to significantly enhance the longevity of the storage network and decrease the duration required for data insertion.

Chiniah et al. [18] conveys the given introduction that conventional cloud systems are encountering difficulties in managing the growing volume of data within the contemporary distributed application landscape. Distributed storage systems, such as those offered by Microsoft, Google, and Amazon Azure, have gained widespread adoption as a means of data storage. Replication is commonly employed as a redundancy strategy. The paper proposes that erasure coding may serve as a feasible substitute. The aim of this paper is to evaluate different distributed storage methods and analyze the potential integration of erasure coding within them. The article provides an overview of established distributed storage systems, taking into account various factors such as availability, consistency, tolerance for parts, principles of design, model data, failure detection and rehabilitation, consistency, and security. The implementation of erasure coding at these types of systems is being analyzed, emphasizing its benefits and identifying unresolved issues. The conclusion highlights that the paper offers a thorough examination for researchers in the domain, delving into the ways in which the integration of erasure codes can augment the functionalities in distributed storage systems.

Guo et al. [27] presents a deduplication scheme that effectively tackles the difficulties associated with encryption on the client and flexible the title management in cloud storage. The scheme employs cryptography using elliptic curves (ECC) to facilitate key sharing among multiple data owners, without the need for intermediaries. The system utilizes broadcast encryption to facilitate ownership management, enabling the cloud service provider (CSP), to regulate users' data access by updating the public key in an efficient manner. The authors have provided a comprehensive overview of their proposed scheme, which includes a detailed system model, analysis of security, and efficiency assessments. One of the key features of the scheme is its ability to facilitate two-party interactive deduplication. The proposed scheme facilitates secure deduplication by enabling direct interaction between the uploader and the cloud service provider, thereby eliminating the requirement for any third-party involvement. This approach can enhance the security of the system by making it more resilient against brute-force dictionary breaches and also helps in minimizing system overhead. One potential approach to managing ownership is through the use of broadcast encryption. The scheme utilizes broadcast encryption to handle the title of outsourced data. The cloud service provider has a process in place to create a list of users for each data, and utilizes a broadcast key to re-encrypt a portion of the ciphertext, thereby ensuring both forward and backward security of the data. This approach offers unique advantages compared to other ownership management methods such as key-encrypting key (KEK) trees while ciphertext-policy attribute-driven encryption (CP-ABE).

Feng et al. [36] discusses the difficulty of costly update overhead in erasure codes in this research study, which focuses on multi-block updates in heterogeneous clusters. In order to efficiently prevent congested connection bottlenecks, the authors suggest a unique approach called Multi-block Double Tree Update (MDTUpdate), which builds a double tree structure for updated data blocks and parity blocks. They use a hybrid update strategy that combines data-delta and parity-delta approaches inside the double tree structure to reduce update costs. To expedite tree formation and improve transmission pathways, a time-efficient greedy method is used. According to experimental findings, Multi-block Double Tree Update greatly outperforms previous approaches in terms of update speed, increasing it by up to 83.23% while keeping a little operating cost.

Levitin et al. [29] presents a unique method for assessing a production-dual storage system's mission success probability (MSP). In order to increase the MSP, it emphasizes speeding up storage unit uploads and downloads. The system demand, the duration of the mission, and the likelihoods that the production unit and storage units would fail are all taken into account by the suggested probabilistic model. In order to demonstrate the model and examine the impacts of different system factors, two case studies of water delivery systems are done. By including component loading effects, the report closes a gap in earlier studies and offers suggestions for improving the efficiency of production-storage systems.

Shin et al. [33] improves the performance of distributed file systems that store huge data by the cache-based matrix approach presented in this study. The method decreases needless cost and boosts performance by using cache memory to store and reuse matrices created during encoding and decoding operations. As a consequence of the design's use of the Weighting Size and Cost Replacement Policy (WSCRP) algorithm, write, read, and recovery times are decreased. The usefulness of the approach is shown by experimental findings utilizing the Hadoop Distributed File System (HDFS) [19] with Reed-Solomon coding, which show shorter recovery times in cases of node failures than conventional HDFS systems.

Chouhan et al. [13] investigates the application of erasure encoding techniques in storage systems, concentrating on the best encoding settings to balance reliability and storage cost while taking user preferences into account. In terms of storage overhead, data accessibility, retrievability, and storage effectiveness, the authors examine the Reed-Solomon coding scheme. They do tests to figure out the best encoding parameter values, then they analyze the results and emphasize how crucial it is to choose the right settings for more dependability and less expensive storage.

Abebe et al. [8] investigates the application of erasure encoding techniques in storage systems, concentrating on the best encoding settings to balance reliability and storage cost while taking user preferences into account. In terms of storage overhead, data accessibility, retrievability, and storage effectiveness, the authors examine the Reed-Solomon coding scheme. They do tests to figure out the best encoding parameter values, then they analyze the results and emphasize how crucial it is to choose the right settings for more dependability and less expensive storage.

Cheng et al. [38] designs a stream machine learning system called StreamLEC which addresses the requirement for fault tolerance in massively dispersed deployments. Erasure coding allows StreamLEC to provide low-redundancy proactive fault tolerance, allowing quick failure recovery while minimizing recovery overhead and preserving low latency. Evaluations on a local cluster and Amazon EC2 show that it supports a variety of stream machine learning applications and outperforms proactive fault tolerance and reactive fault tolerance in terms of performance. Overall, StreamLEC offers a practical approach to stream machine learning fault tolerance, enhancing performance and guaranteeing quick recovery.

Kulkarni et al. [7] highlights the object storage capabilities of OpenStack Swift as they address the difficulties of rising data storage needs in cloud storage systems. Erasure coding is suggested as a dynamic strategy to boost storage effectiveness while preserving dependability, availability, and fault tolerance. Erasure coding disperses data over several places, as opposed to conventional replication techniques, which lowers storage overhead. The article explains the OpenStack Swift erasure coding policy and underlines the significance of modifying storage methods to satisfy the expanding data storage needs in cloud settings.

In comparison to conventional replication techniques, Nachiappan et al. [40] address the use of erasure coding in data centers in clouds for high dependability with little overhead. They do, however, draw attention to the fact that information recovery in erasure codes might use up a lot of network resources. In order to solve this problem, proactive recovery algorithms that choose data blocks for replicating based on failure predictions have been developed. These algorithms often fall short of optimizing the selection process, however. The authors provide a recovery strategy to address this issue that reduces the amount of data blocks chosen for proactive replication. They do this by taking into account acceptable and essential limits created utilizing the system's existing network activity and data duplication information. The trade-off involving energy use and bandwidth reduction during proactive recovery is also highlighted by the authors, since extra temporary storage cost may offset the energy benefit from reduced bandwidth utilization. The authors calculate the energy use of battery packs in order to examine energy usage. Through extensive simulations, they assess the suggested algorithm and contrast it with a heuristic proactively recovery strategy. According to the experimental findings, the suggested recovery method decreases storage cost by 46% and network usage by 60% when compared to the suggested heuristic technique. In addition, compared to replication, the suggested proactive recovery approaches may save up to 52% of the energy used by storage systems.

Facenda et al. [26] researches on the implementation of adaptive streaming codes within a network scenario involving three nodes. The network consists of a source node, or no an intermediary, and an end node. The relay facilitates the transmission of a sequence of packets containing messages from the source to the destination. The source-to-relay along with relay-to-destination links may experience occasional packet erasures, with a maximum limit of N1 and N2, respectively. The main goal is to guarantee that the destination receives every message packet within a specific time frame T. The author presents a novel approach to streaming codes which are customized to different parameters, including N1, N2, and T. This adaptation is based on the erasure patterns discovered from the original data set to the relay. This can be accomplished by utilizing symbol estimates, which enable the relay to transmit information regarding symbols prior to their decoding. Furthermore, variable-rate encoding is utilized, which decreases the rate of packet encoding as it encounters more erasures.

Qin et al. [16] makes a suggestion on how to enhance reading and writing efficiency in the erasure coding mode of the Ceph system by selecting storage nodes that take resource load status and node heterogeneity into account. The authors recommend using Ceph erasure coding in a heterogeneous combination storage method. To thoroughly assess storage node activities, they take into account a variety of performance indicators, such as CPU consumption and storage space use. The storage node with the greatest performance is identified using the TOPSIS approach, and is then given Ceph erasure coding workloads. Using node diversity and resource load status to optimize task allocation and overall efficiency, experimental findings show that this strategy improves read/write performance for big items in Ceph's erasure coding mode.

Iliadis et al. [28] researches on erasure-coding redundancy techniques in large-scale storage systems to guard against device failures which is covered in this overview. With the use of closed-form formulas for the metrics Mean Time to Data Loss (MTTDL) and Expected Annual Fraction of Data Loss (EAFDL), a theoretical model is created to examine the effects of hidden sector defects. The research includes distributions for bit error rates, data insertion patterns, and device breakdown and rebuild times. The results demonstrate that although EAFDL is mostly unaffected for modest erasure codes, MTTDL suffers with actual sector error rates, but it may also deteriorate for strong codes. Superior dependability is provided by the declustered data placement strategy. In conclusion, this study offers perceptions into the dependability of erasure-coded storage systems, highlighting the need of resolving hidden faults and taking into account the best data placement techniques.

Arslan et al. [25] provides an introduction to reliability theory and concentrates on the prediction of availability and durability of erasure-coded storage systems in warm/cold storage settings. It discusses basic ideas, enhanced stochastic models, and the author's innovations, which include the invention of a broad kind of Markov model for calculating mean time to failure. Comparing storage setups, exploring multi-dimensional Markov models for detachable drive-medium combinations, and making the case for their applicability to upcoming DNA storage libraries are all covered in this article. As a means of overcoming the drawbacks of both simple and complicated Markov models, simulation modeling is also covered. It accepts that debates are still pertinent for systems repaired one device at a time even if the focus is on simultaneously maintained systems.

The use of erasure coding to increase dependability while lowering overhead in cloud data centers is covered in the article. When compared to replication, erasure coding data recovery uses a lot of network bandwidth. Calheiros et al. [40] provides a recovery technique that optimizes data block selection for proactive replication by taking into account limitations based on the system's network traffic and data duplication information. In comparison to a heuristic proactive recovery strategy, the suggested algorithm decreases storage overhead by 46% and network traffic by 60% via extensive simulations. Furthermore, compared to replication, proactive recovery techniques may save the storage system up to 52% of its energy. The paper offers an effective method that, via proactive replication that is tuned, increases dependability while minimizing network bandwidth use, storage overhead, and energy consumption in cloud data centers.

Song et al. [34] approaches an adaptive hybrid storage which dubbed FACHS (File Access Characteristics-based Hybrid Storage) is suggested to deal with these problems. Concerns around storage redundancy, data availability, and durability have surfaced with the growth of distributed systems and big data technologies. Traditional multi-copy storage techniques provide an excessive amount of storage redundancy, and current erasure coding techniques ignore file access characteristics, leading to subpar parallel read/write performance and squandered storage capacity. Low computational and storage costs are ensured by FACHS' use of RS Code for cold files with infrequent use. Multi-copy and LRC codes, which improve efficiency and parallel read/write capabilities while lowering recovery costs, are used for small and big hot files, respectively. According to the findings of the experiments, FACHS increases hot file recovery efficiency by 29%, read/write speed by 8%, and cold file storage space occupancy by 12%. FACHS maximizes performance and storage effectiveness by taking file access factors into account.

Rivera et al.[32] discusses the necessity for dependable multicast services in heterogeneous networks with emphasis on Network Coding (NC) across data networks. The need for non-feedback multicast techniques and the dearth of dynamic coding control in present communication systems are brought to light. The study suggests the Fulcrum coding transmission technique as a solution to these problems. To increase data transmission reliability and flexibility, this approach combines random linear network coding (RLNC) with systematic coding. The suggested solution seeks to improve network performance, decrease transmission times per packet, and boost decoding probability by adopting efficient Forward Error Correction (FEC).

Pu et al. [31] discusses how Ultra-High-Definition (UHD) videos are becoming more and more popular and how difficult it is to broadcast them owing to their high bandwidth needs. By using in-network video streaming, UHD video quality may be improved with the introduction of 5G networks that are cloud native. On in-network servers, a bottleneck may be caused by a lack of storage or bandwidth. The authors suggest a brand-new UHD video streaming system called EMS that combines multi-source streaming with erasure-coded storage. In addition to proposing a federated learning strategy for adaptive service quality upgrades, they propose measures to gauge the effectiveness of video servers. Additionally, they tackle the issue of user local training while preserving streaming Quality-of-Experience (QoE) by redefining it as a Multi-Armed Bandit (MAB) problem. They construct a prototype of EMS and create Upper Confidence Bound (UCB)-based algorithms with theoretical guarantees, proving the efficacy of the latter via thorough testing.

After going through all these studies, we can say that various storage systems were used to implement data using various EC policies and EC schemes; however, we were unable to find any papers that were linked to analyse time efficiency using benchmark data that we have performed.Here, it shows a comparison between all our remarkable findings and our own work -

Reference	Purpose of the Study	Technology used	Brief of Findings
Wang et al. [22]	Compare DSS	Erasure coding	DLOP-partitioning-based storage scheme
Tanwar et al. [21]	Data partitioning and deployment	Distributed Linear Order Partition (DLOP)	DLOP and partitioning-based storage strategy
Lombardo et al. [29]	RAID storage frequent parity data updates	All-flash arrays, redundant parity data	Reducing parity data write operations compared to standard data deduplication
Guo et al. [26]	Cloud encryption and title management	Elliptic curve cryptography (ECC)	Direct uploader-cloud service provider deduplication improves system security and eliminates third-party intervention
Arslan et al. [24]	Erasure-coded warm/cold storage systems' availability and endurance	Stochastic models, Markov models	detachable drive-medium Markov models
Levitin et al. [28]	Assessing production-dual storage system mission success probability (MSP)	Probabilistic model for MSP	Optimizing production-storage systems
Nandyal et al. [15]	DFS data availability and stability	Replication and erasure coding	EC saving space over replication
Siagian et al. [17]	Video surveillance database system architecture network performance improvement	OpenStack	Openstack in IaaS
Our Work	Time efficiency and Fault Tolerance measurement	Reed Solomon, Openstack Swift, Erasure Coding	Less time efficient, More fault tolerant

Chapter 3

Background

3.1 Overview of Openstack and Erasure Coding

The OpenStack component is responsible for provisioning object storage services. Swift technology has been developed to offer a robust and flexible solution for the storage of vast quantities of unstructured data, with a focus on scalability and durability. The data is stored within designated "containers", which can be structured hierarchically and retrieved through an API. Swift facilitates the storage of vast amounts of data and employs a ring-based framework that facilitates data distribution across numerous servers in a manner that is efficient and resilient to faults. Swift offers data durability, ensuring data protection against hardware failures, network failures, and other forms of data loss. The use of replication and erasure coding accomplishes this by ensuring multiple copies of the data are stored securely on separate servers. In general, the utilisation of distributed and fault-tolerant systems is prevalent in various industries, including media and entertainment, healthcare, and scientific research. These systems are particularly useful for managing substantial volumes of data in a distributed manner.

Swift and other distributed storage systems use Erasure Coding (EC) as a technique of data security to prevent data loss due to hardware malfunctions or other sorts of loss. EC involves breaking down data into smaller fragments and including additional parity fragments to enhance data redundancy [35]. The fragments are distributed among various storage nodes within an atomic system for storage. In the event of a node failure in the storage system, the system has the capacity to utilise the number of parity pieces to restore the original data. EC offers significant benefit in terms of data protection by utilising less redundancy compared to conventional methods such as RAID [1]. Erasure coding offers scalability , fault tolerance and defence from double failures, whereas standard RAID methods are designed to protect against one disk failure. Erasure coding has gained significant popularity in distributed storage technologies such as Swift, as it plays a crucial role in safeguarding data against failures in the hardware and other forms of data loss in extensive storage environments. The use of EC can provide significant benefits in terms of both data protection and efficiency.

In a commonly used erasure coding technique known as Reed-Solomon coding, the data is partitioned into blocks, and supplementary parity blocks are created based

on the initial data blocks. The quantity of parity blocks produced is determined by the desired level of protection.
After generating the data and parity fragments, they are strategically distributed across several storage nodes to optimise both efficiency and fault tolerance. The storage system utilises a fragmentation process to distribute data across multiple storage nodes. In the event of a failure, the system has the capability to utilise the remaining fragments in order to reconstruct the initial data. This process appears to be quite efficient and has a higher tolerance for errors compared to traditional data protection methods.

In erasure coding, for constructing parity pieces firstly, data is split up into blocks. The first thing that has to be done in order to begin the process of erasure coding is to split up information into blocks that are all the same size. It's possible that the data will be segmented into chunks that are each 1 megabyte in size.
Once the information has been segmented into blocks,the required degree of data security determines the maximum number of parity pieces that may be created. Calculating the parity fragments requires utilising a mathematical function that takes as input the informational blocks that go into making up the stripe. The erasure coding method plays a role in determining the particular function that is applied. In the case of Reed-Solomon Coding algorithm is what determines how the process of constructing parity pieces in erasure coding is carried out. Whenever the parity pieces have been formed, they are spread among numerous storage nodes in a manner that optimises both fault tolerance and efficiency.

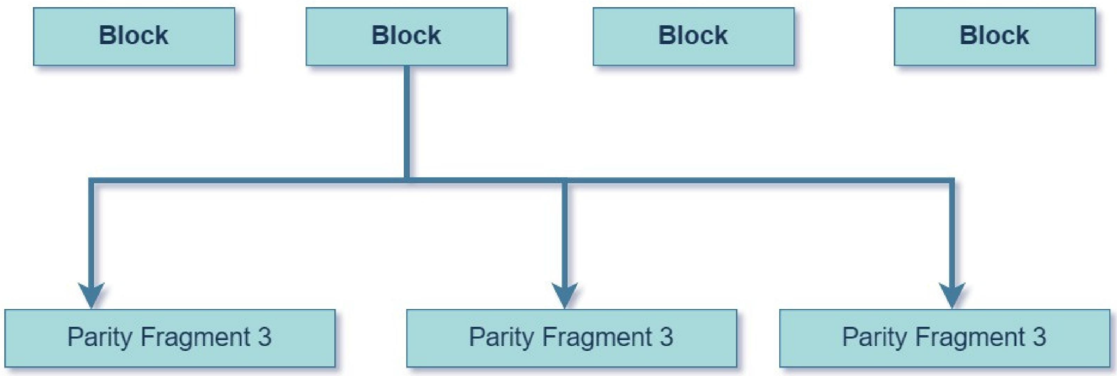

Figure 3.1: Parity Fragments in Erasure Coding.

Example blocks contain three parity pieces. Security and erasure coding determine parity pieces. Fault tolerance demands more parity. Each block needs two parity pieces to recover from two offline storage nodes. Parity fragments demand more storage. Distributed storage systems use erasure coding to recover lost data. The procedure involves data separation and parity creation. Data and parity are dispersed between storage nodes. The system can recreate the original data if node failures or other difficulties destroy pieces. Data integrity requires parity. Parity fragments allow cloud computing data recovery.

Erasure coding protects OpenStack Swift data. Swift object storage leverages data fragment parity. Data and parity segments separate the cluster's storage nodes. When accessing or retrieving the item, the system may reassemble data and parity pieces. Erasure coding redundancy safeguards Swift data against intentional loss or alteration. Swift restricts access and encrypts data. Erasure coding and other OpenStack Swift security measures safeguard sensitive data.

3.2 Erasure Coding Techniques

a) Reed-Solomon codes : The widespread use of Reed-Solomon codes in cloud storage systems is a result of their potent error-correcting abilities. They are widely utilized in Swift installations and provide a good balance between encoding and decoding effectiveness.

b) Locally Repairable Codes (LRC): LRCs are a subset of erasure codes that are designed to cut down on the amount of repair bandwidth required during data recovery operations. By limiting the quantity of data sent over the network, LRC offers fault tolerance while streamlining the repair procedure.

c) Fountain codes: To provide effective data encoding and decoding, fountain codes combine random linear data chunks. They have a reputation for being resilient to packet loss and provide benefits in terms of encoding speed and adaptability when managing various storage capacities.

d) Piggybacking codes: These codes use the redundant information created by erasure coding to provide extra data or metadata along with the encoded data. With less distinct information communication overhead, this method intends to improve fault tolerance.

e) Turbo codes: Turbo codes are iterative error correcting codes that use iterative decoding methods to provide great dependability. These codes have shown potential for enhancing the efficiency and overall effectiveness of erasure coding methods.

These erasure coding methods were chosen to showcase a variety of coding schemes, error-correcting capabilities, and optimization methodologies. We can learn more about these methods' advantages, disadvantages, and applicability to the Swift cloud storage system by analyzing them.

3.3 Reed-Solomon (RS) for Swift

Reed-Solomon (RS) codes have been widely used in cloud storage systems, including the Swift object storage system. Reed-Solomon codes offer a robust and efficient approach to achieving fault tolerance and data integrity in distributed storage environments. This brief section provides an overview of how Reed-Solomon codes are employed in the Swift system. In the Swift cloud storage system, Reed-Solomon codes are utilized for data protection and recovery purposes. When a file is uploaded to Swift, it is divided into data chunks, and additional parity chunks are computed using the Reed-Solomon coding technique. The number of data and parity chunks is determined based on the configured coding parameters. The Reed-Solomon coding scheme used in Swift allows for the reconstruction of the original file even if a certain number of chunks are lost or become unavailable. This fault tolerance property makes Reed-Solomon codes particularly suitable for cloud storage systems where data durability and reliability are crucial. During file retrieval, Swift retrieves the necessary data and parity chunks from the underlying storage nodes and employs the Reed-Solomon decoding algorithm to reconstruct the original file. The decoding process involves performing mathematical operations on the available chunks to recover any missing or corrupted data. One advantage of Reed-Solomon codes in Swift is their efficiency in terms of storage space utilization. By employing Reed-Solomon codes, Swift can minimize the overhead associated with data redundancy while providing the desired level of fault tolerance. This efficiency is achieved by carefully selecting the number of data and parity chunks based on the desired fault tolerance requirements.

3.4 Impact of the Data Deduplication and Erasure Coding in OpenStack Swift's performance

It is possible to consider utilizing data deduplication in conjunction with erasure coding to potentially improve performance within OpenStack. Data deduplication is a useful technique that can help to minimize storage costs and enhance performance by storing only unique data and eliminating any duplicates.

When data deduplication is combined with erasure coding, it can effectively minimize the volume of data that requires encoding and storage as parity fragments. This approach has the potential to enhance the speed of encoding and decoding processes, while also minimizing the network bandwidth needed for data transfer.

It is worth considering that the implementation of data deduplication may result in some increase in processing power and memory usage. Hence, it is crucial to thoroughly assess the advantages and disadvantages of utilizing data deduplication alongside erasure coding in a particular OpenStack implementation.

3.5 Simulators for Erasure Coding

To emulate erasure coding in OpenStack, a variety of simulators are available. A few of the well-liked simulators are:

1. COSbench is an open-source cloud storage storage performance benchmarking tool. It might be used to mimic the OpenStack erasure coding process.

2. Cloudsim is a popular cloud computing simulation program. It can be used to mimic how erasure coding works in OpenStack [2].

3. Mininet: This free, open-source network emulator can be used to mimic how OpenStack's erasure coding functions in practice [9].

4. OpenStack simulator: The OpenStack software itself comes with a simulator that can be used to test how well erasure coding works in OpenStack.

By adjusting the settings, such as the quantity of data and parity pieces, block size, data redundancy, etc. These simulators can be used to mimic various erasure coding scenarios.

3.6 SimEDC: A Simulator for Evaluating Erasure Coding in Cloud Storage Systems

SimEDC [12] is a Python-based simulator for assessing the performance of distributed storage systems that use erasure coding. Users can customize different erasure coding schemes and system settings.

To assess system performance in various settings, it generates synthetic workload and takes into account many characteristics. Because of the simulator's flexibility and extensibility, it is simple to add new erasure coding schemes and alter existing ones.

In order to effectively utilize simEDC, it is necessary to configure the simulation tool with a range of parameters and inputs that accurately reflect the unique characteristics of the data center becoming simulated. The parameters that need to be considered include the topology of the data center, erasure codes, placement of redundancy, and breakdown patterns of various subsystems.

It is provided with specific details for the simulation, including the overall number of iterations, the total number of processes to be utilized, and the designated mission time (in hours) for the simulation to run. Additionally, we can consider setting various parameters that are pertinent to the situation, such as the number of stands, nodes per rack, disks per node, capability per disk, chunk size, number of horizontal stripes, and the entire capacity. Furthermore, we carefully selected the code type, such as Reed-Solomon, and determined what was needed for n (number of chunks) along with k (number of parity chunks). The installation type for the redundant was determined to be "FLAT", which means that it is distributed equally across the data center. The user provided specific details regarding the system's network locations failure designs, and simulation type. The authors employed the simulation type of uniformization_balanced_failure_biasing and thoughtfully included the probabilities and beta values for value sampling.

After ensuring that all parameters have been properly configured, it is necessary to initiate the simulation employing SIMEDC. The simulator proceeded to run the simulation using the designated inputs and produced the corresponding outcomes. The simulation's output comprised of several reliability metrics, including the Probability of Data Loss (PDL), Repair Efficiency (RE), Number of Megabytes of Data Loss (NOMDL), Bandwidth Reduction (BR), and Single-chunk repair ratio.

Through the implementation of multiple simulation runs with diverse configurations, the authors conducted an analysis and evaluation of the dependability of erasure-coded network preservation across a range of scenarios and conditions.

3.7 Data processing in Openstack with Hardware acceleration

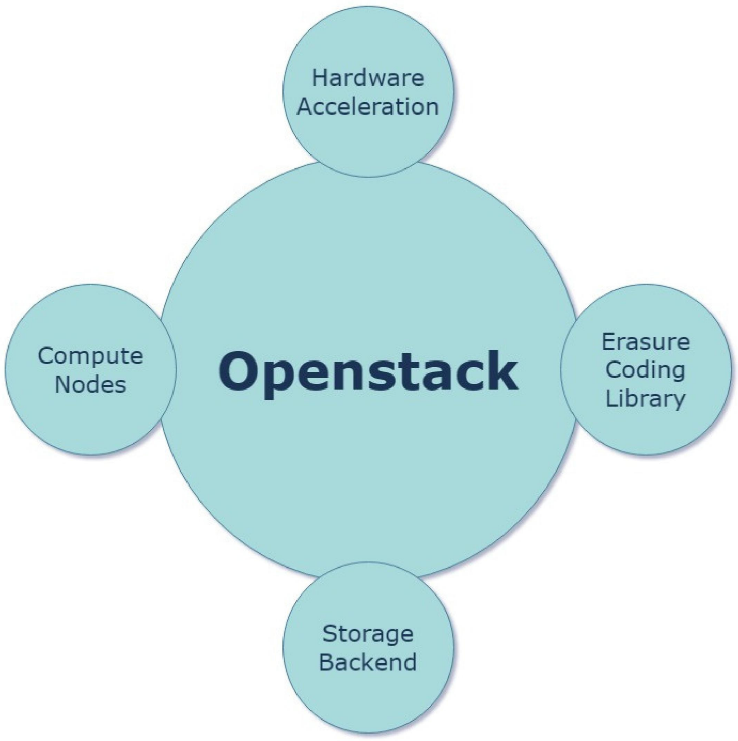

Figure 3.2: Accelerating Data Processing in OpenStack.

In this diagram, the hardware acceleration is represented by a distinct hardware component that is mounted on the compute nodes, such as an FPGA. For some operations, such matrix multiplication, the erasure coding library is changed to make use of the hardware acceleration device, which can considerably quicken the processing of multiplication. The compute nodes can outsource some processing duties to the hardware device by using hardware acceleration, which can lead to faster erasure coding operations and less overhead in terms of processing power and memory utilization.

There are numerous phases involved in data deduplication in OpenStack Swift utilizing erasure coding:

1. Data is initially split into fixed-size units known as data fragments.

2. Next, erasure coding is used to divide each data fragment into a number of parity fragments.

3. The parity and data fragments that are produced are kept in the storage cluster.

4. The system determines whether each data fragment already exists in the storage cluster before saving it.

5. If the data fragment is already there, the system makes a pointer to the current copy rather than saving it again.

6. This pointer is kept in a different mapping or index database.

7. The system looks for the location of the data in the index or mapping database when a client requests a data fragment.

8. The system retrieves the data and gives it to the client if it is already existent in the storage cluster.

9. If the data is missing, the system gets the required data and parity fragments, decodes them, and then sends the client the original data.

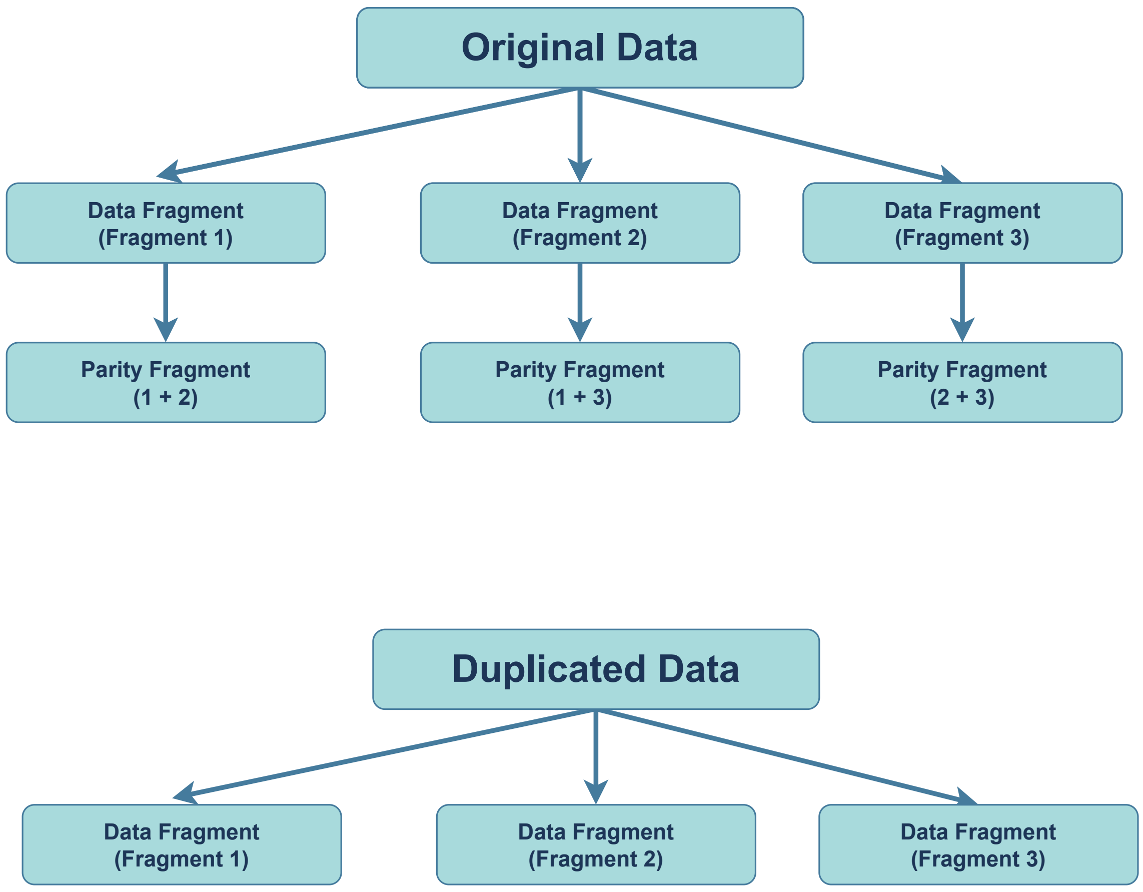
Original Data
Data Fragment (Fragment 1)
Data Fragment (Fragment 2)
Data Fragment (Fragment 3)
Parity Fragment (1 + 2)
Parity Fragment (1 + 3)
Parity Fragment (2 + 3)
Duplicated Data
Data Fragment (Fragment 1)
Data Fragment (Fragment 2)
Data Fragment (Fragment 3)

Figure 3.3: Parity Fragments in Erasure Coding

In this illustration, the original data is divided into three fragments (Fragment 1, Fragment 2, and Fragment 3), and three parity fragments are produced using erasure coding. In the event of failures, the parity pieces are then used to retrieve any lost data.

The data fragments are then subjected to data deduplication before being stored in OpenStack. Data deduplication finds redundant data and removes it, keeping only one copy of each distinct data fragment. This lowers the amount of data storage space required, which lowers storage expenses.

One copy of each unique data fragment is stored in the deduplicated data, which significantly reduces the amount of storage space needed. For this deduplicated data, parity fragments can be created using the same erasure coding process, guaranteeing data availability and integrity.

Chapter 4

Methodology

4.1 Research Work-plan

In our work, we want to experiment on different dataset for benchmarking EC schemes. In order to do so firstly, we plan to select a Suitable EC Technique for our later experiments. Secondly, we create an Experimental Test bed for benchmarking EC Schemas. Then we select Datasets which is followed by creating our Own dataset and using an existing one. Thirdly, our method follows different implementations for our desired results ie. analyzing data.

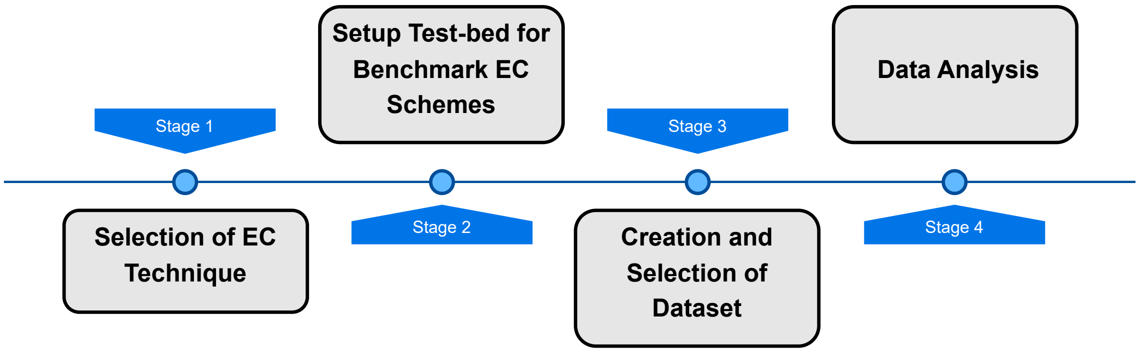
Stage 1
Selection of EC Technique
Setup Test-bed for Benchmark EC Schemes
Stage 2
Stage 3
Creation and Selection of Dataset
Data Analysis
Stage 4

Figure 4.1: Workflow.

4.1.1 Selection from Existing Erasure Coding Techniques

In our work, we prioritize the selection of suitable erasure coding techniques for assessing time efficiency and fault tolerance properties in Swift cloud storage systems. To ensure a fair and comprehensive evaluation, we follow a specific methodology and criteria for choosing erasure coding schemes. These criteria include considering the popularity and extensive use of approaches within the cloud storage domain, especially in the context of Swift, to ensure proven effectiveness. Compatibility with Swift infrastructure is also crucial, favoring methods explicitly designed or modified for seamless integration. We prioritize newly developed or significantly improved techniques to incorporate the latest advancements in erasure coding research. Additionally, we select a diverse range of coding schemes, such as Reed-Solomon codes and Luby Transform codes, to provide a thorough examination of Swift's erasure coding capabilities. By adhering to these selection criteria, we aim to ensure a relevant and reliable assessment of erasure coding algorithms in the context of Swift cloud storage systems.

However, we selected Reed Solomon code as our EC technique . Reed-Solomon codes play a vital role in the Swift cloud storage system, providing fault tolerance and data integrity. By leveraging Reed-Solomon codes, Swift ensures that data remains recoverable even in the face of node failures or data corruption. The use of Reed-Solomon codes in Swift demonstrates the system's commitment to maintaining high availability, durability, and reliability for cloud-based object storage.

4.1.2 Methods for setup the testbed Environment

For setting up a test bed environment we set up both Remote and Local test bed environments. For both test beds after the swift installation process in local and remote Virtual Machines (VM), we configure the Erasure Coding (EC) Schemes . We work on three EC schemes for our work which are RS (5+3) , RS (7+5) and RS (10+4). RS stands for Reed Solomon code which has been used as EC technique for our Cloud Storage System Swift.

Methods of EC-Setup for Local Testbed

In this subsubsection, we describe the methodology employed for the EC (Erasure Coding) setup within the local VMs running OpenStack Swift. The process involves connecting to the Swift server, authenticating as an admin, configuring storage policies, building Swift rings, and creating EC containers for storing data. After establishing a connection to the Swift server and authenticating as an admin, the EC setup process was initiated within the local VM. This ensure administrative access and the ability to perform the necessary configuration tasks. To begin the EC setup, the first major task was to define the storage policy. This involved specifying the desired parameters and settings to establish the appropriate EC policy for data storage. Different EC policies were required for each schema being evaluated in the thesis project. It is worth noting that it is possible to establish different storage policies within a single VM, allowing for the execution of multiple schemas simultaneously. This aspect will be discussed further in a later part of this thesis.

Subsequently, Swift rings were built for each storage policy. These rings determined the distribution and placement of data across the available storage partitions. Each schema was associated with a unique storage policy and corresponding Swift ring. Throughout the EC setup process, three different local VMs were utilized. Swift was installed on each VM, followed by the configuration of the storage policy specific to the corresponding EC schema. As a result, each VM represented a distinct EC schema, with its own Swift setup and associated storage policy. Data upload and evaluation were conducted on the different EC schemas to assess the performance of storing data under varying EC policies. This allowed for a comparative analysis of the efficiency and effectiveness of different EC setups in terms of data storage and retrieval.

Methods of EC-Setup for Remote Testbed

In this subsubsection, we detail the methodology employed for accessing and configuring a remote Virtual Machine (VM) to expand the evaluation of the cloud storage system. After configuring the local VM and working with datasets to analyze the performance, the need arose to explore the behavior of the system in a remote VM environment. To accomplish this, a remote VM was procured from a cloud company specializing in private cloud solutions. Upon connecting to the remote VM, the Swift installation process was replicated, mirroring the steps followed for the SAIO VM (local VM) configuration. By installing Swift on the remote VM, it was ensured that the same cloud storage system was available for evaluation. As the thesis project involved multiple schemas of erasure coding, a challenge arose due to having only one remote VM available. To overcome this limitation, multiple storage policies were configured within a single remote VM. This allowed for the exploration of different erasure coding schemas within a unified remote VM environment. Configuring multiple storage policies within the remote VM involved defining the necessary parameters and settings for each policy. This enabled the project to work with multiple erasure coding schemes within a single remote VM, facilitating comparative analysis and evaluation. To accommodate the diverse storage policies, different Swift rings and erasure coding containers were created. Each storage policy required its own set of Swift rings to determine data distribution and placement. Additionally, separate erasure coding containers were established for each policy to store and manage data.

Once the creation of containers under multiple erasure coding policies was successfully completed, the project proceeded with the implementation of the dataset on these schemes. This allowed for an assessment of the performance and behavior of the cloud storage system when operating on different erasure coding Schemes within the remote VM environment.

Simulator for Testbed:

For benchmark Erasure Coding (EC) Schemes we are measuring results based on time efficiency and fault tolerance of Erasure Coding. For our testbed we are also adding a simulator (SimEDC) in our testbed to figure out the fault tolerance of EC. To assess system performance in various settings, SimEDC generates synthetic workload and takes into account many characteristics. Because of the simulator's flexibility and extensibility. Additionally, it is simple to add new erasure coding schemes and alter existing ones. The simulation's output comprised of several reliability metrics, including the Probability of Data Loss (PDL), Repair Efficiency (RE), Number of Megabytes of Data Loss (NOMDL), Bandwidth Reduction (BR), and Single-chunk repair ratio.

4.1.3 Datasets for Testbed

As our analysis demands to work with not only with different sizes of data but also a large number of files in order to benchmark the Erasure Coding Schemes thoroughly. For making our needs mitigated we have created our own dataset to perform data analysis. The dataset we created MCSD-100 (details at 5.1.1) not only has a range of files we need for data analysis, it also consists of 4 different types of files which includes text, image , video and audio files. Secondly, for mitigate the need of using vast number of files , we are using a benchmark dataset name COCO-17 (details at 5.1.2) .

4.1.4 Approaches for Data Analysis

For our desired result we analyse dataset with different approach. For instance, in order to check time efficiency of erasure coding we try to do upload data on cloud based storage system Swift. We then download and delete files from swift. For each functions we measure time to upload, download and delete time. We repeat this method for three EC Schemes which are based on Reed Solomon (5+3), (7+5), (10+4). We try this functionalities both for both Remote and Local Testbed. The testbed also includes simulator SimEDC which gives us the result for fault tolerance of EC coding for various EC Schemes.

Chapter 5

Datasets

5.1 Datasets

In our work, we are utilizing two distinct datasets, namely the MCSD (Multitype Cloud Storage Dataset) [39] and Coco-17 [4] (Common Objects in Context-17). The MCSD is a self-created dataset specifically designed to evaluate the performance of the Swift cloud storage system with various file types and sizes. On the other hand, Coco-17 serves as a widely recognized benchmark dataset widely used for object detection and image understanding tasks. By employing these datasets, we aim to explore and analyze the behavior and capabilities of cloud storage systems in handling diverse data types and compare their performance against established standards. On our local and remote servers, we utilized both of these datasets.

5.1.1 Creating a Benchmark Testing Dataset for Object Storage (MCSD-100)

Our work presents a self-created dataset aimed at evaluating the performance of the Swift cloud storage system. Our objective is to assess how Swift handles data storage, retrieval, and maintenance, particularly in relation to its EC (Erasure Coding) policy and fragmentation capabilities. However, finding an existing dataset that met our specific criteria proved challenging. We required a dataset encompassing a diverse range of data types and sizes, ranging from 100KB to 1GB. To address this gap, we curated a comprehensive dataset consisting of text, audio, image, and video files, representative of real-world data. These files were carefully selected to mirror typical file sizes encountered in cloud storage scenarios. By examining the behavior of each file type and size under various EC schemas, we can provide valuable insights into the suitability and efficiency of Swift's storage mechanisms. This dataset serves as a valuable resource for researchers and practitioners interested in cloud storage systems. Its availability allows for reproducibility and facilitates future studies to build upon our findings. In our work, we describe the dataset creation process, provide detailed information on the included file types and sizes, and present the experimental setup for evaluating Swift's performance. The results of our analysis contribute to a better understanding of cloud storage performance and offer recommendations for optimizing the utilization of Swift in practical deployments.

So as per its Description, let's name it "Multitype Cloud Storage Dataset (MCSD-

100): A Comprehensive Collection of 100 Text, Audio, Image, and Video Files."

Data Collection:

The data collection process involves employing various approaches to ensure the inclusion of diverse file types and sizes in our dataset. To obtain video files with a wide range of sizes, we captured videos of different lengths using mobile devices. This approach allows us to generate a distribution of video file sizes that closely resembled real-world scenarios. In addition to video files, we have sourced pictures from our personal mobile devices to incorporate image files in the dataset. To achieve the desired sizes for the image files, we have utilized online converters, enabling us to obtain JPEG files of specific sizes. By employing these data collection strategies, we are ensuring that our dataset comprised a comprehensive range of file types and sizes, closely mirroring the characteristics of data commonly encountered in cloud storage environments. This approach enhances the representativeness and applicability of our dataset, allowing for more accurate assessments of Swift's performance under diverse file size distributions.

File Sizes:

The dataset comprises 100 files, distributed across four different file types: text, mp3, jpeg, and mp4. Each file type exhibits distinct size characteristics, which are summarized below:

1. Text Files (in KB): The 25 text files present in the dataset exhibit sizes ranging from 102 KB to 4786 KB. The average size of the text files is 2092.6 KB, with the smallest file measuring 102 KB and the largest file measuring 4786 KB. The distribution of text file sizes provides a representation of common document sizes encountered in practical cloud storage scenarios.

2. Mp3 Files (in KB): The 25 mp3 files span a range of sizes from 101 KB to 20357 KB. On average, the mp3 files in the dataset have a size of 10176.96 KB. The smallest mp3 file has a size of 101 KB, while the largest mp3 file measures 20357 KB. The inclusion of mp3 files of various sizes allows for a comprehensive evaluation of Swift's handling of audio files, capturing typical file size variations in this format.

3. Jpeg Files (in KB): Our dataset includes 25 jpeg files, with sizes ranging from 644 KB to 95832 KB. The average size of the jpeg files is 36399.28 KB, with the smallest file measuring 644 KB and the largest file measuring 36399.28 KB. The diverse range of jpeg file sizes reflects the variability in image file sizes commonly encountered in cloud storage scenarios.

4. Mp4 Files (in KB): The 25 mp4 files encompass sizes ranging from 1000 KB to 1015100 KB. On average, the mp4 files in the dataset have a size of 399460 KB. The smallest mp4 file has a size of 1000 KB, while the largest mp4 file measures 1015100 KB. The inclusion of mp4 files with different sizes enables a comprehensive examination of Swift's performance in storing and retrieving video files.

By incorporating files of varying sizes within each file type, our dataset encompasses a wide range of file size distributions. This enables a robust evaluation of Swift's performance across different EC schemas and fragmentation techniques, providing valuable insights into the behavior of the cloud storage system under real-world conditions.

Our comparison and analysis of the self-created dataset revealed valuable insights into the performance of the Swift cloud storage system across different file types and sizes. Swift demonstrated efficient storage and retrieval of text files, robust handling of audio and video files, and reliable preservation and distribution of image files. These findings have implications for enterprise data management, multimedia content platforms, image hosting and sharing, long-term data preservation, and collaborative content creation. Swift's capabilities in handling diverse file types and sizes make it a versatile solution for various data storage and retrieval needs. Overall, our analysis indicates that Swift performs admirably in storing and retrieving files across various types and sizes. The system showcases robust capabilities in managing text, audio, image, and video data, demonstrating its suitability for diverse cloud storage requirements.

In the subsequent sections, we will discuss the implications of our findings and provide recommendations for optimizing Swift's performance based on the observed behaviors and performance characteristics across different file types and sizes.

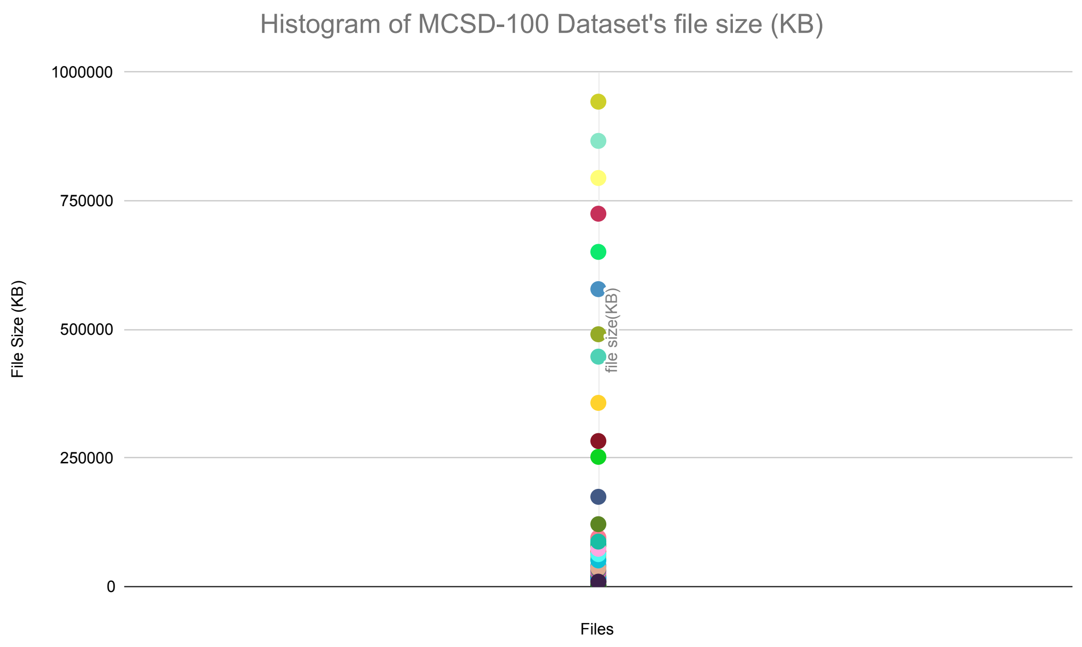

Figure 5.1: Histogram of MCSD-100 Dataset's file size.

Potential Applications: The findings from our research indicate several potential applications for the Swift cloud storage system. These include enterprise data management, multimedia content distribution, image hosting and sharing, data archiving and long-term preservation, and collaborative content creation. Swift's efficient handling of text, audio, image, and video files makes it suitable for storing and retrieving diverse data types. Organizations can leverage Swift for managing textual data, distributing multimedia content, hosting and sharing images, preserving valuable data over the long term, and facilitating collaborative content creation. These applications highlight the versatility and adaptability of Swift in meeting various data storage and retrieval needs.

Availability and Access:

To ensure the broader accessibility and usability of our dataset, we have made it publicly available for download. Researchers, practitioners, and interested individuals can access and retrieve the dataset from the following link: [39]. By providing open access to our dataset, we aim to facilitate further research, experimentation, and collaboration in the field of cloud storage and data management.

5.1.2 Another Dataset: COCO-17 (Common Objects in Context-17)

The COCO-17 [4] dataset, created primarily for object identification and picture interpretation tasks, is a widely used benchmark in the area of computer vision. It consists of a vast array of annotated photos covering 17 different item categories, providing a varied and rich dataset for analysis and assessment. The dataset is carefully selected to include circumstances and situations that occur in the real world, assuring its relevance and suitability for use in a variety of computer vision applications. It includes photos taken in a variety of settings, such as interior and outdoor situations, and under varying lighting and vantage points. COCO-17 offers accurate and comprehensive annotations for 17 popular item types. These categories include a broad variety of things, including people, animals, cars, and everyday household goods. The annotations provide useful data for training and assessing object identification and instance segmentation algorithms. They comprise object bounding boxes, segmentation masks, and keypoints. The COCO-17 dataset is big in that it includes several pictures and their related annotations. This scale facilitates thorough examination and assessment of computer vision models and methods. The dataset has also evolved into a common benchmark for academics and practitioners, promoting cooperation and enabling accurate performance comparisons across various methodologies. The COCO-17 dataset, which offers a uniform baseline for assessing object identification, instance segmentation, and picture interpretation tasks, has been significant in developing the area of computer vision. Modern algorithms have been developed as a result, and it has been widely used in research articles, challenges, and contests. We used a subset of the COCO-17 dataset made up of 3000 JPEG photos for our particular investigation. To provide a representative sample for our study goals, this subset was chosen, enabling us to concentrate on certain elements while using the variety and richness of the overall dataset. In conclusion, the COCO-17 dataset presents an extensive collection of pictures with in-depth descriptions, including 17 different item categories. Its widespread use and meticulous annotation procedure make it a useful tool for analyzing and comparing computer vision algorithms, advancing the field's knowledge of object identification and picture interpretation.

Dataset Size and Subset Size:

The widely used COCO-17 dataset for computer vision contains a big collection of pictures and comments. The overall number of pictures in the COCO-17 dataset is significant, totaling to [enter total number of photos]. These photos have been painstakingly annotated with specific details like object bounding boxes, segmentation masks, and keypoints.

We made sure that our selection had exactly 3000 photos by defining'max_samples=3000'. This strategy enabled us to concentrate our research while taking into account the practical restrictions of processing resources and time constraints on a subset that properly reflected the variety and richness of the whole COCO-17 dataset.

In conclusion, we used the FiftyOne library to download 3000 pictures from the COCO-17 dataset for our thesis study, specifying the'max_samples=3000' option. This carefully chosen subset, which was based on certain object types, served as the basis for our investigation. We were able to address our research goals while efficiently managing computing resources and time restrictions by using this representative subset, which allowed us to conduct a focused and insightful analysis that fit within the parameters of our study.

Importance of COCO-17 in Our Research Work:

Due to a number of crucial reasons, the COCO-17 dataset must be included in our thesis study. First off, compared to our test dataset, MCSD, the COCO-17 dataset provides a substantially higher size and diversity of files. This dataset offers a rare chance to investigate the behavior and traits of object storage systems, notably Swift, while dealing with a large number of files since it has a wide range of file sizes, ranging from 0 KB to 600 KB. It is impossible to overstate the difficulties we faced while gathering the information for such a vast number of files. It took a lot of work and careful preparation to gather and organize the COCO-17 dataset's various file kinds and sizes. In order to ensure their inclusion across different size distributions, the technique entailed acquiring a broad variety of picture files. Through this project, we were able to extensively examine and analyze the interactions between various file kinds and sizes and the underlying storage infrastructure and assess how well object storage regulations, including EC schemas, handled these data changes. By using the COCO-17 dataset, we were able to get beyond the constraints placed on us by our smaller test dataset, MCSD, and we were able to learn a great deal about the behavior and efficiency of object storage systems. The addition of this dataset in our study improves the validity and dependability of our conclusions since it provides a larger, more varied collection of data that more accurately reflects actual situations.

Overall, by including the COCO-17 dataset in our thesis study, we have been able to broaden the breadth and depth of our analysis and investigate the potential and difficulties that come with handling a large number of files of various sizes. We want to advance the area of data management and storage in large-scale contexts by advancing our knowledge of and improvements to object storage systems like Swift.

Chapter 6

Experimental Evaluation

6.1 Experimental Setup

Experimental setup consists of testbed environment setup, setup dataset for testbed and testbed simulation implementation. For testbed environment after installing Swift for both local and remote testbed we setup EC-coding for both environments. Then for setup dataset for testbed, we use some code to upload , download and delete data on Swift and then analyse the results from the output.

6.1.1 Testbed Environment Setup:

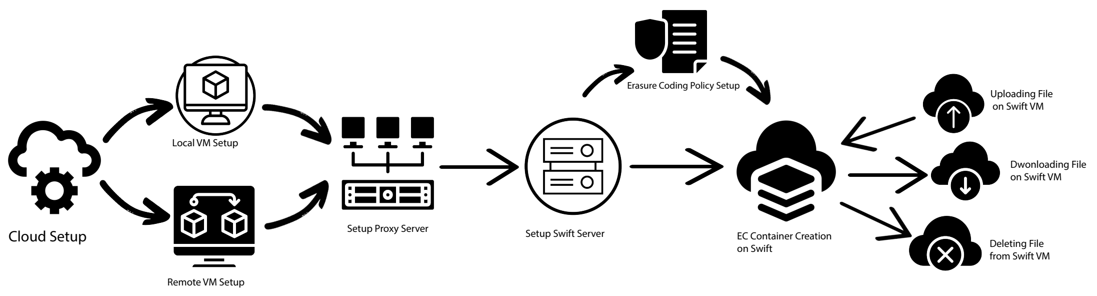
Cloud Setup
Local VM Setup
Remote VM Setup
Setup Proxy Server
Setup Swift Server
Erasure Coding Policy Setup
EC Container Creation on Swift
Uploading File on Swift VM
Dwonloading File on Swift VM
Deleting File from Swift VM

Figure 6.1: Test bed Environment Setup

EC-setup for Local Testbed:

To see how well erasure coding saves time, we need to break up the data into different rules. To do this, we must first install Swift and set up erasure coding by using different EC strategies. The following is how EC is set up:
After installing swift on our local computer and making sure it works, we need to set up EC strategy in swift. To set up an EC policy in Swift, we made a policy description file that lists the settings for the EC method. Most of the time, the policy description file is in the /etc/swift folder on the Swift proxy server. The erasure

coding strategy says how many data and parity pieces there will be and where it will go on the storage nodes. To define EC policy we need to change the value of names policy_type, ec_type, ec_num_data_fragments, ec_num_parity_fragments, ec_object_segment_size. For both schemas "5 + 3" and "7 + 5" we set:

```
policy_type = erasure coding,
ec_type = liberasurecode_rs_vand
ec_object_segment_size = 1048576
ec_num_data_fragments = n
ec_num_parity_fragments = m
```

Here , 'n' and 'm' will be changed according with schemas. For example, for '5+3' schema, the value of n will be 5 and the value of m will be 3 .

After creating the policy definition files for the EC policies, we can use them to build a ring. A ring is a configuration file that describes the layout of the Swift cluster, including the number of nodes, their IP addresses, and their assigned roles (e.g., proxy server, storage server, etc.). We created a ring for both of the policies using this swift command-

```
swift-ring-builder <ringfile> create <part_power> <replicas>
<min_part_hours>.
```

The Swift ring builder builds and handles the linking of the data and parity pieces to the storage nodes. This makes sure that the fragments are spread out evenly across the cluster. The Swift ring maker also figures out how many copies of the data should be made and sent to different parts of the cluster. In erasure coding, replicas are not exact copies of the data. Instead, these are extra pieces of data that are made using the same math method as parity fragments. The Swift ring maker makes sure that the right number of copies are made and spread across the cluster so that data is always available. After the fast ring builder makes the ring, the ring file needs to be sent to all of the storage nodes in the cluster. This has been done using the swift-ring-builder command,

```
<ringfile> rebalance
```

Which distributes the ring file to all of the nodes and ensures that the data and parity fragments are evenly distributed across the nodes. As soon as our cluster was ready to take in data, we prepared our data by breaking it up into pieces and making the right parity pieces. Once the data is ready, it can be sent to the Swift object store system using the normal calls to the Swift API.

EC-setup for Remote Testbed:

In order to achieve superior analytical results, we implemented our benchmark data (Coco 17) [4] and our own dataset on a remote server in addition to our local server. We could access our remote server via a network connection by establishing an SSH connection. Secure Shell (SSH) is a network protocol that provides secure access to a remote host using cryptography. We used the "ssh" command followed by the username and the server's IP address or domain name to establish an SSH connection to a remote server (ssh username@server-ip-address). We employed a password-based authentication method to establish connections with remote servers. After connecting via SSH, we can access the remote server's command-line interface. Through the command-line interface, swift and erasure coding is configured within the swift configuration. We set up EC on a remote server like the way we described before.

Each server on our local server independently implements a unique EC policy. Nevertheless, a single server is responsible for storing data using various EC policies on the remote server. The primary advantage of this method is its centralized storage and management. Utilizing a single remote server optimizes the use of hardware resources, such as CPU and storage space. The server can allocate resources dynamically based on the requirements of various EC policies. The various EC policies applied to a single remote server were 10+4, 7+5, and 5+3.In order to configure three different erasure coding (EC) policies in the Swift configuration file (swift.conf), we modified the [storage-policy: default] section. In the swift configuration file (swift.conf), within the [storage-policy: default] section, we define the three distinct EC policies with distinct identities. Each policy specifies comparable values for,
ec_type, policy_type, and ec_object_segment_size but unique values for
ec_num_data_fragments and ec_num_parity_fragments. For a 10+4 policy,
ec_num_data_fragments=10 and ec_num_parityfragments=4 are specified. The same holds true for 7+5 and 5+3. Swift will recognize the three EC policies defined in the [storage-policy: default] [41] section following these modifications. Using the Swift API or command-line tools, we can now designate these policies to specific storage containers or objects, allowing us to select the desired EC policy for each item of data based on our needs.

6.1.2 Setup Dataset for Testbed

Our main objective is to investigate how different data kinds behave in accordance with different EC regulations depending on upload, download, and deletion timeframes. After establishing EC policies in our swift server, we created a swift ring based on them. Swift ring is built to choose the locations for data storage inside the Swift storage cluster. A request to store the data is made, and the ring is examined to select the appropriate storage nodes for the data. Then, we create an AUTH token so that we may access the Swift storage service for authentication and authorisation. To generate Auth token we use the command -

```
curl -v -H 'X-Auth-User: <username>' -H 'X-Auth-Key: <password>
<auth_url>
```

We may establish an ec-container to manage and organize our data after producing the Auth token. To build containers-

```
curl -v -X PUT -H ‘X-Auth-Token: <auth_token>’ \<auth_url>" -H
"X-Storage-Policy: <ec_policy>" <swift_url>/<container_name>
```

We generated three python script files to analyze data based on upload, download, and delete times. By using the command line to launch the Python3 interpreter, we can execute our Python files on local and distant VMs.

Uploading Data on Swift:

The following code is used to upload files to Swift and calculate how long it takes. The directory of the specific folder we want to work on was provided, and the code uploaded and timed each file in that folder.

Algorithm 1 File Upload Algorithm

```
admin $\leftarrow$ "AUTH_admin"
auth_token $\leftarrow$ 'AUTH_tk675285d84f05430389bf74dbbaa14e6c'
container_name $\leftarrow$ 'ec-container'
url $\leftarrow$ 'http://127.0.0.1:8080/v1.0/{}/{}'.$format$(admin, container_name)
headers $\leftarrow$ {'X-Auth-Token' : auth_token}
folder_path $\leftarrow$ '/home/upom/Documents/coco'
file_paths $\leftarrow$ an empty list
for all file in the list of files in folder_path do
    if file is a file (not a directory) then
        append os.path.join(folder_path, file) to file_paths
    end if
end for
for all file_path in file_paths do
    container_file_name $\leftarrow$ basename of file_path
    open file_path as f in read binary mode
    start_time $\leftarrow$ current time
    make PUT request to '{}/{}'.$format$(url, container_file_name) with data = f and headers = headers
    end_time $\leftarrow$ current time
    if status_code of the response is 201 then
        upload_time $\leftarrow$ end_time $-$ start_time
        display "File {} uploaded successfully. Upload time: { .2f} seconds."
.format(container_file_name, upload_time)
    else
        display "File {} upload failed with status code: {}"
.format(container_file_name, status_code of the response)
    end if
end for
```

This code assumes that the file we wish to upload resides on our local system in the "/home/upom/Desktop/coco" folder. The local_folder_path variable will need to be updated to reflect the path of the folder from which we want our files to upload. Additionally, we need to confirm that the Swift instance's Swift endpoint URL and authentication token are appropriately configured. The time.time() method is being used to determine the beginning and ending timings for each file upload. The total upload time is then determined by subtracting the two times, and it is included in the output message for each file that has been uploaded. The UPLOAD request will return a 404 status code if the requested file name cannot be located in the container. We display an error message in this instance to inform the user that the file could not be uploaded.

To further clarify, in order to automate the system, add the module "time" before the upload command prompt. If we also want to know the system time, user time, and actual time it takes to upload a file, the code will look like this:

Algorithm 2 File Upload Algorithm

```
admin ← "AUTH_admin"
auth_token ← 'AUTH_tk52bab9c40d22465bb856cec97cce0a77'
container_name ← 'ec-container'
url ← 'http://127.0.0.1:8080/v1.0/'.format(admin, container_name)
headers ← {'X-Auth-Token' : auth_token}
display "Enter a list of file paths to upload, separated by commas: "
read file_paths_str
file_paths ← split file_paths_str by commas
for all file_path in file_paths do
    strip leading/trailing whitespace from file_path
    container_file_name ← basename of file_path
    upload_command ← 'time curl -v -X PUT -H "X-Auth-Token: {}" {} -T ""' .format(auth_token, '/'
.format(url, container_file_name), file_path)
    display upload_command
    make PUT request to '{}/{}'.format(url, container_file_name) with data = open(file_path, 'rb') and headers = headers
    r ← the response of the request
    if status_code of the response is 201 then
        execute upload_command using a subprocess
    else
        display "File {} upload failed with status code: {}"
.format(container_file_name, status_code of the response)
    end if
end for
```

At upload_command in the code, we inserted a "time" module. A Unix function called time calculates how long it takes a command or program to execute. We may get information about the execution time, such as system time, user time, and actual time, in the output by prefixing the curl command with time.

Downloading Data from Swift

The below code is used to download files from Swift and calculate the download time. File names are inputted into this code, which then deletes the files in accordance with the Swift server.

Algorithm 3 File Download Algorithm

```
    admin ← "AUTH_admin"
    auth_token ← 'AUTH_tk8ec2188bf76e45cd8b5ef90ed2d5998e'
 3: container_name ← 'ec-container'
    url ← 'http://127.0.0.1:8080/v1.0/{}/{}'.$format$(admin, container_name)
    headers ← {'X-Auth-Token' : auth_token}
 6: display "Enter the file names to download (separated by commas):"
    read file_names_str
    file_names ← split file_names_str by commas and trim whitespace
 9: local_file_path ← '/home/upom/Desktop'
    for all file_name in file_names do
        start_time ← current time
12:     make GET request to '{}/{}'.$format$(url, file_name) with headers = headers
        end_time ← current time
        if status_code of the response is 200 then
15:         open file with path os.path.join(local_file_path, file_name) in write binary mode
            write content of the response to the file
            display
18: "File {} downloaded successfully in { .2f} seconds.".format(file_name, end_time - start_time)
        else
            display "File {} download failed with status code: {}".format
21: (file_name, status_code of the response)
        end if
    end for
```

The "file_names" variable in this code would simply hold a list of file names, and the function would loop over each file name and download each file from Swift. The downloaded files would be stored with identical filenames to the container in the "local_file_path" directory. The start and finish timings of each file download are obtained using the time.time() method. The total download time is then determined by subtracting the two times, and it is included in the output message for each file that has been downloaded. The Download request will return a 404 status code if the requested file name cannot be located in the container. In this instance, we produce an error message to inform the user that they were unable to download the file.

Deleting Data from Swift:

Using the following code, it may remove files from Swift and calculate how long it takes to do so. File names are inputted into this code, which then deletes the files in accordance with the Swift server.

Algorithm 4 File Deletion Algorithm

```
admin ← "AUTH_admin"
auth_token ← 'AUTH_tk8ec2188bf76e45cd8b5ef90ed2d5998e'
container_name ← 'ec-container'
url ← 'http://127.0.0.1:8080/v1.0/{}/{}'.$format$(admin, container_name)
headers ← {'X-Auth-Token' : auth_token}
display "Enter the file names to delete (separated by commas): "
read file_names_str
file_names ← split file_names_str by commas and trim whitespace
for all file_name in file_names do
    start_time ← current time
    make DELETE request to '{}/{}'.$format$(url, file_name) with headers = headers
    end_time ← current time
    delete_time ← end_time − start_time
    if status_code of the response is 204 then
        display "File {} deleted successfully in { .2f} seconds." .format(file_name, delete_time)
    else
        display "File {} delete failed with status code: {}" .format(file_name, status_code of the response)
    end if
end for
```

For each file we wish to remove, a remove request is sent to Swift using the 'requests.delete()' method in this code. In order to confirm that the file was successfully erased, we are also looking at the status code of the response. The DELETE request will return a 404 status code if the requested file name cannot be located in the container. For the user's information, we display an error notice in this scenario to indicate that the file cannot be erased.

We link our Python file with our Swift cluster using the URL of the code. The aforementioned algorithms enable us to generate our projected data results on both local and distant servers based on time.

6.1.3 Setup Simulator for Testbed: SimEDC

The SIMEDC simulator is a discrete-event simulator that has been specifically designed to analyze the reliability of erasure-coded data center storage. Please find below a brief outline of the operational process of SIMEDC.

In the configuration phase, the user defines the simulation elements and inputs. The aforementioned factors encompass the data center topology, erasure codes, redundancy duty, failure themes, and other pertinent configurations. The parameters that can be considered include the number of racks, the number of nodes per rack, the quantity of disks per node, the capacity per disk, the size of each chunk, the type of code used (such as Reed-Solomon), and the number of data chunks and parity chunks.

The simulation environment is initialized by SIMEDC in accordance with the specified configuration. The process involves configuring the data center topology, storage equipment, failing structures, and other relevant elements of the simulated system.

The simulation's event scheduling is managed through an event queue by the simulator. Events may encompass data read and write requests, system failures, repairs, and other related occurrences. The scheduling of events is based on their respective occurrence times.

SIMEDC follows a chronological order to process events from the event queue. The system manages various events by executing their respective actions and updating the simulation's state accordingly. In the event of a failure, the simulator designates the affected component as inaccessible and initiates a repair event.

SIMEDC employs repair and redundancy techniques to address system failures. In the event of data loss or unavailability, the system simulates the repair process by initiating data reconstruction. The system computes the data recovery by utilizing the erasure codes and redundant placement parameters specified in the arrangement. The simulation tool also monitors the maintenance activities and their influence on the system's dependability.

Data Loss Probability, Repair Efficiency, Number of Megabytes Lost, Bandwidth Reduced, and Other Measures are Collected During Simulation by SIMEDC. The aforementioned metrics offer valuable insights regarding both the efficiency and dependability of the erasure-coded storage system utilized in the data center.

The simulation will execute events until it meets a predetermined termination criterion, such as a designated mission duration or the fulfillment of a specific number of iterations.

Analysis of Results: Upon completion of the simulation, SIMEDC furnishes the analysis results based on the gathered metrics. The aforementioned outcomes provide valuable information regarding the dependability, effectiveness, and functionality of the erasure-coded storage system in a data center setting, as per the given configuration.

The aforementioned process provides a comprehensive overview of the functioning of SIMEDC. The simulator provides the flexibility to configure inputs and customize simulations, thereby facilitating researchers to assess and appraise the dependability of erasure-coded information preservation under diverse scenarios and conditions.

6.2 Experimental Result

6.2.1 Graph Analysis

Here we made an analysis of upload time on remote and local VM of coco data set and also we have used 100 data sets to determine upload download and delete time.

Benchmark COCO-17 Dataset Analysis Result for Local and Remote Testbed

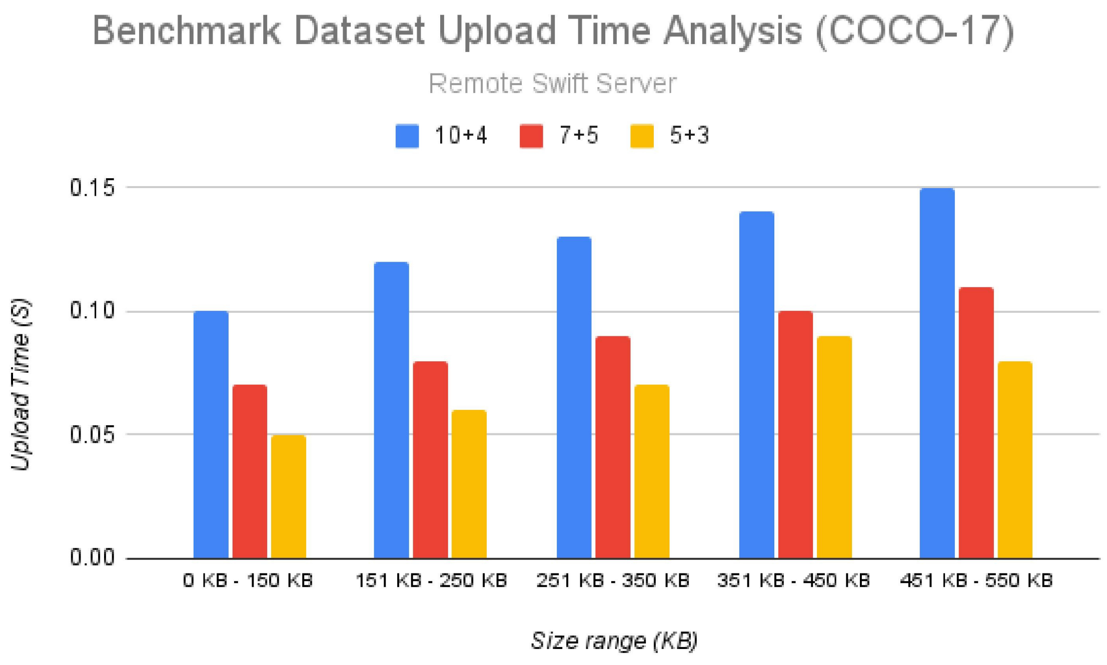

Figure 6.2: Benchmark Dataset(coco17-remote swift server).

The graph from FIgure 6.2 illustrates a benchmark dataset upload Time analysis (COCO-17) [4] on the basis of a remote swift server. Here the x-axis demonstrates upload time in second and the y-axis shows size range expressed in KB. The blue bar shows data fragmentation of 10+4 whereas the red and yellow bar represents data fragmentation of 7+5 & data fragmentation of 5+3 respectively. Within the range 0 KB to 150 KB The 10+4 bar takes the longest upload time in comparison with red and yellow. Within this range the upload time for blue bar is 0.10 s and the red and yellow bar takes nearly 0.07 and 0.05 seconds. The same relationship between the bars are easily unravelled within the other ranges ; For example from 151 KB-250 KB upto 451 KB-550 KB.That means,within the limit of 0KB to 550 KB size range 10+4 data fragmentation takes the prolonged time and the time gradually increases with the increasing data size ranges. The same way 7+5 data fragmentation takes the shortest time compared to 10+4. The data fragmentation of 5+3 takes the shortest time assimilating with that two. Moreover, 7+5 and 5+3 also increase following the same manner as the blue with growing size ranges.So, the result of overall representation of the graph is showing the relationship of upload time with size range of data where the 10+5 data fragmentation takes the prolonged time & the 5+3 data fragmentation takes the shortest time in comparison among them.

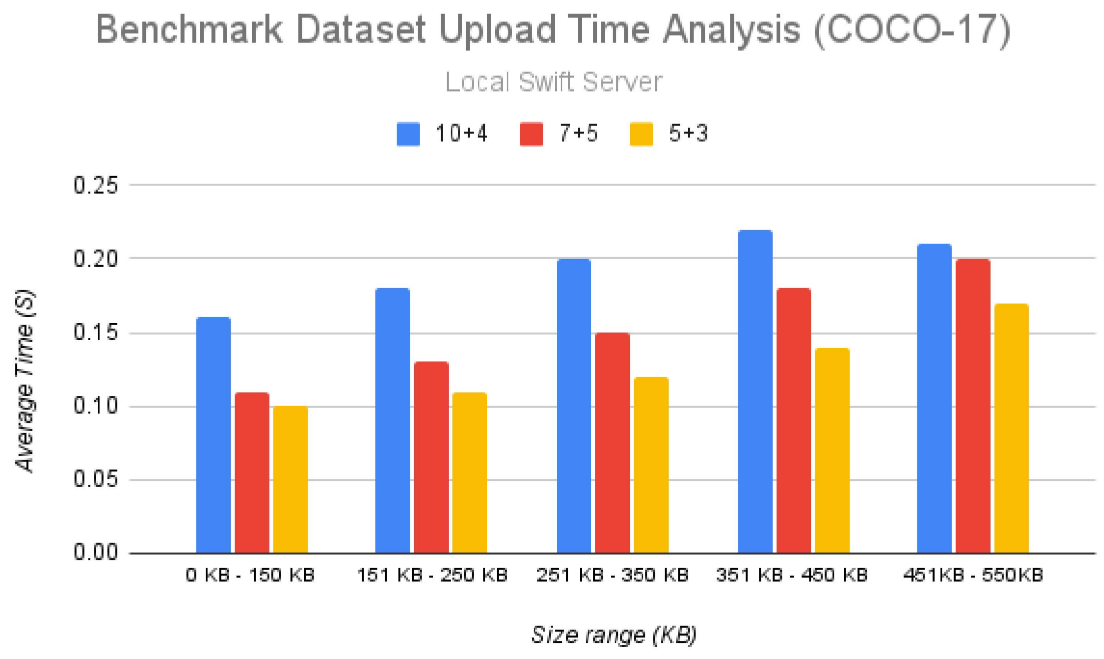

Figure 6.3: Benchmark Dataset(coco17- local swift server).

The graph from Figure 6.3 demonstrates a benchmark dataset upload Time analysis (COCO-17) on the basis of local swift server. Here the x-axis shows average time in second and the y-axis shows size range expressed in KB. The red and yellow bars reflect data fragmentation of 7+5 and 5+3, while the blue bar depicts data fragmentation of 10+4. Within the range 0 KB to 150 KB The 10+4 bar takes the longest upload time in comparison with red and yellow. Within this range the upload time for the blue bar is nearly 0.16 s and the red and yellow bar takes nearly 0.11 and 0.10 seconds. The same relationship between the bars are easily unravelled within the other ranges ; For example from 151 KB-250 KB upto 451 KB-550 KB. But the differences of the average time for each of the types of data fragments gradually mitigates as they reach the range of 451KB-550 KB. That means,within the limit of 0Kb to 550 KB size range the 10+4 data fragmentation takes the prolonged time and the time gradually increases with the increasing data size ranges upto the limit of 450 KB. When the size range increases from 451 KB upto 550 KB the time for 10+4 data fragment somewhat decreases. From the previous range that means within 351KB-450 KB but the relationship will be same for the other two types. Besides this, 7+5 data fragmentation takes the shortest time compared to 10+4. The 5+3 data fragmentation takes the shortest time assimilating with that two. Moreover, the 7+5 & 5+3 also increases following the growing size ranges. So, the result of the overall representation of the graph is showing the relationship of the average time with size range of data where the 10+4 data fragmentation takes the prolonged time upto the range of 450 KB & the time slightly decreases in 451-550 KB size range & the 5+3 data fragments takes the shortest time in comparison among them.

Benchmark Testing Dataset (MCSD-100) Analysis Result for Local Testbed

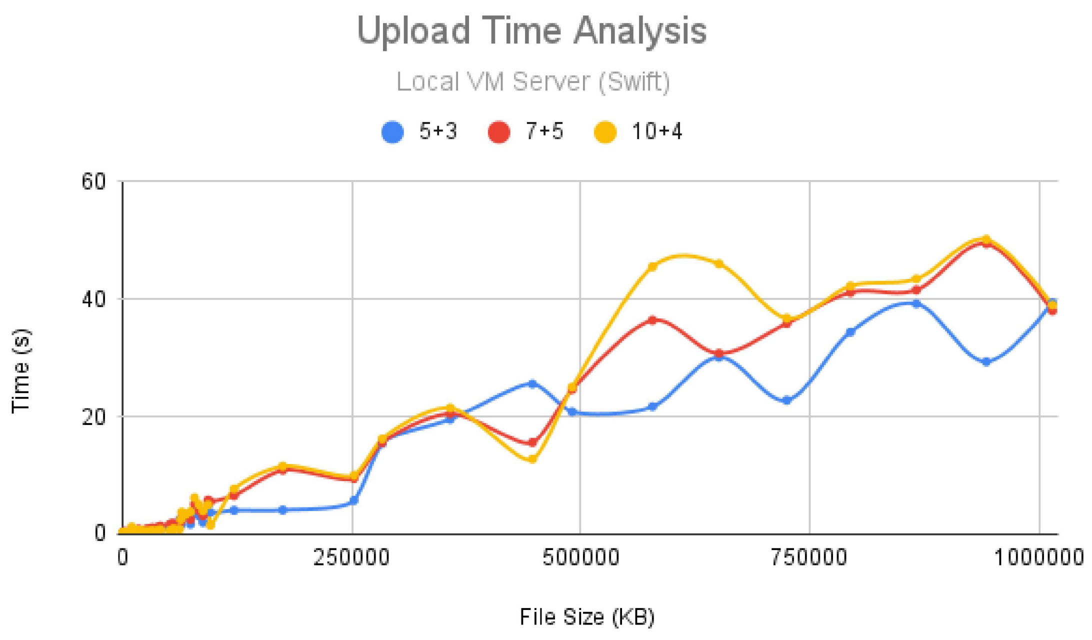

Figure 6.4: Upload time analysis

The line graph from Figure 6.4 compares the upload times for our self made dataset MCSD-100 [39] ranging in size from 100 KB to 1000000 KB on a local server for the 5+3, 7+5, & 10+4 EC policies. Here the x-axis shows average time in second and the y-axis shows size range expressed in KB. Each data point shows the amount of time needed to upload a particular size of files. As we can see, the upload time lengthens for the 5+3, 7+5, and 10+4 EC policies as the data file size does. However, if we contrast the line graph for the three policies shown in the graph, it is evident that higher fragmentation results in a longer upload time on Swift.

After examining the graph showing the upload time on a local server, we discovered that the upload times for 5+3, 7+5, and 10+4 are, respectively, 3.831789474 seconds, 4.671368421 seconds and 4.921578947 seconds. The average values we obtained after analysis support the finding that more valuable fragmentation requires a longer upload time than less valuable fragmentation.

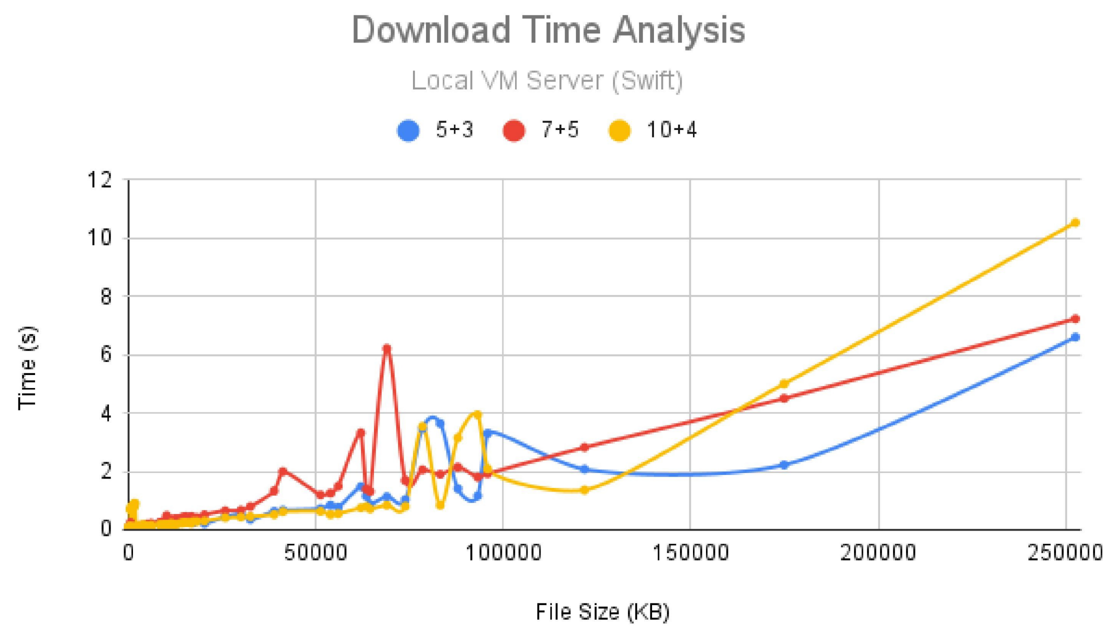

Figure 6.5: Download time analysis

The line graph from Figure 6.5 shows how three erasure coding (EC) policies—5+3, 7+5, and 10+4—differ in terms of download times. The figure includes a range of file sizes from 100 KB to 250,000 KB. Here the x-axis demonstrates download time in second and the y-axis shows size range expressed in KB The graph's data points are connected by straight lines to indicate the overall trend. Each data point on the graph reflects the download time for a particular file size.

The chart demonstrates that as the file size increases, the download time also increases for all three EC policies: 5+3, 7+5, and 10+4. However, upon comparing the line graphs for the three policies, it becomes evident that the policies with higher levels of fragmentation (10+4) require more time to download the files on the swift storage system.

The average download time for the 5+3 policy was determined to be 0.7124418605 seconds, the average download time for the 7+5 policy was 0.9238372093 seconds and the average download time for the 10+4 policy was 0.9439772727 seconds after examining the download time data on the local server. These average figures confirm the finding that download times are longer when there is more fragmentation than when there is less fragmentation.

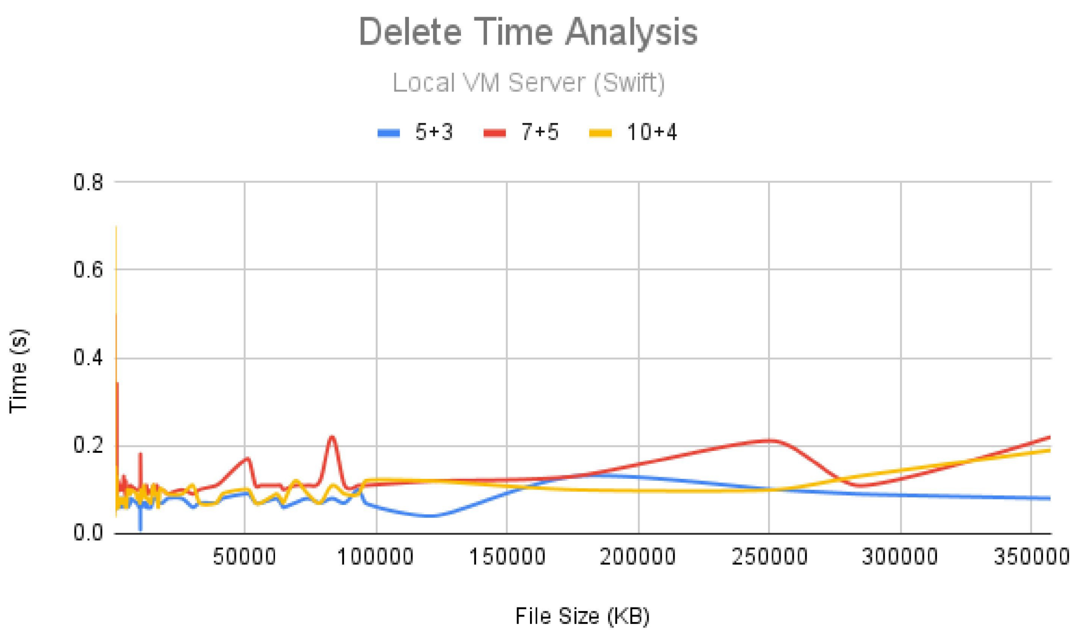

Figure 6.6: Delete time analysis

The line graph from Figure 6.6 shows how three erasure coding (EC) policies—5+3, 7+5, and 10+4—variate in their delete timings. On a local server, the chart shows various file sizes ranging from 100 KB to 350,000 KB. The x-axis displays the file sizes in KB and the y-axis shows the delete time in seconds. Each data point represents the delete time of a certain file size. Examining the data results from the graph, it was found that employing these three erasure coding (EC) policies, deleting files from swift typically takes 0.09515031897 seconds.

Benchmark Testing Dataset (MCSD-100) Analysis Results for Remote Testbed

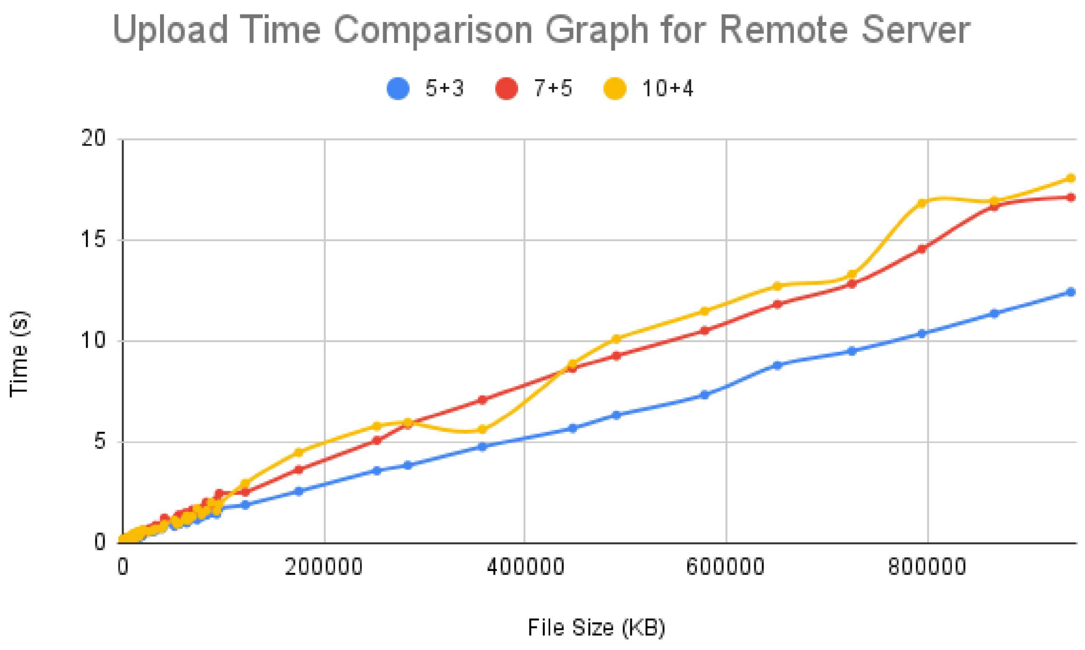

Figure 6.7: Upload time comparison

The line graph from Figure 6.7 above compares the upload times for our self made dataset MCSD-100 [39] ranging in size from 100 KB to 800000 KB on a remote server for the 5+3, 7+5, & 10+4 EC policies. The x-axis displays the file sizes in KB and the y-axis the upload time in seconds. Each data point represents the upload time of a particular file size.

The chart shows that for all three EC policies—5+3, 7+5, and 10+4—the upload time increases as the file size increases. Comparing the line graphs, however, reveals that longer upload times on the swift storage system are a direct result of bigger data fragmentation.

The average upload time for the 5+3 policy was found to be 1.259361702 seconds, the average upload time for the 7+5 policy was 1.76712766 seconds and the average upload time for the 10+4 policy was 1.820425532 seconds after analysis of the upload time data on the remote server. These average values are consistent with the finding that upload times increase when fragmentation levels rise relative to lesser levels.

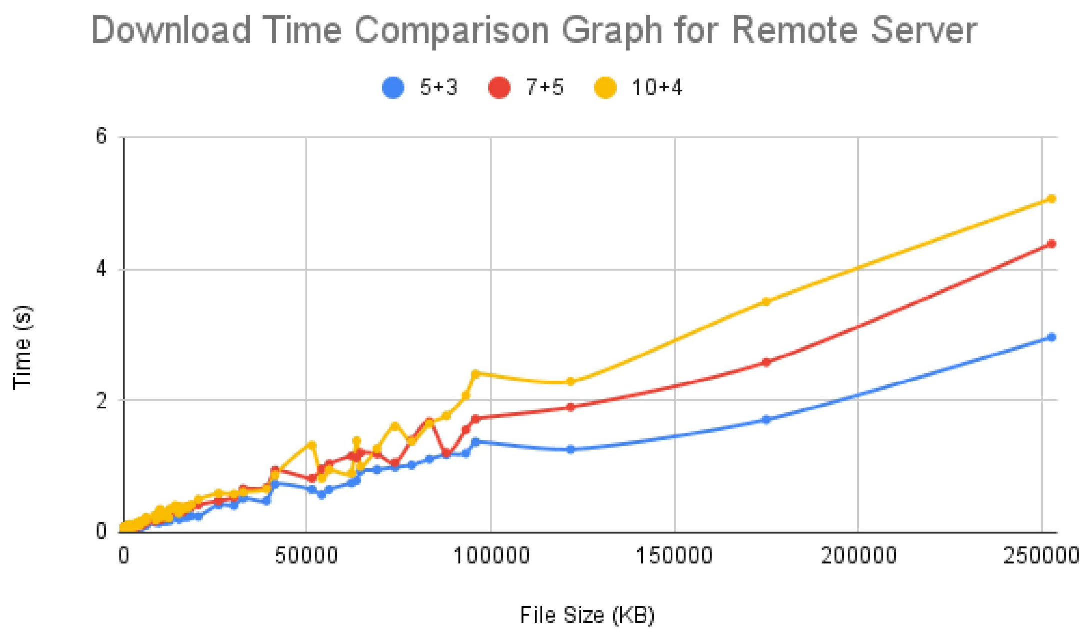

Figure 6.8: Download time comparison

The line graph from Figure 6.8 illustrates the differences in download times for 100 files ranging in size from 100 KB to 250,000 KB on a remote server for three EC policies: 5+3, 7+5, and 10+4. The time is displayed on the y-axis in seconds, while the file sizes are displayed on the x-axis in KB. The data points on the graph are connected by straight lines to show the general trend. The download time for each data point on the graph corresponds to a specific file size.

The graph shows that for all three EC policies—5+3, 7+5, and 10+4—the download time increases as the file size does. In contrast, it is clear from comparing the line graphs for the three policies that the higher levels of fragmentation (10+4) take longer time to download the files.

The average download time for the 5+3 policy was found to be 0.9977659574 seconds after analyzing the data on download times from the remote server. The average download time for the 10+4 policy was calculated to be 1.9839772727 seconds, while the average download time for the 7+5 policy was found to be 1.362021277 seconds. These average results proves that, big data fragmentation take longer time to download files in compare to small data fragment.

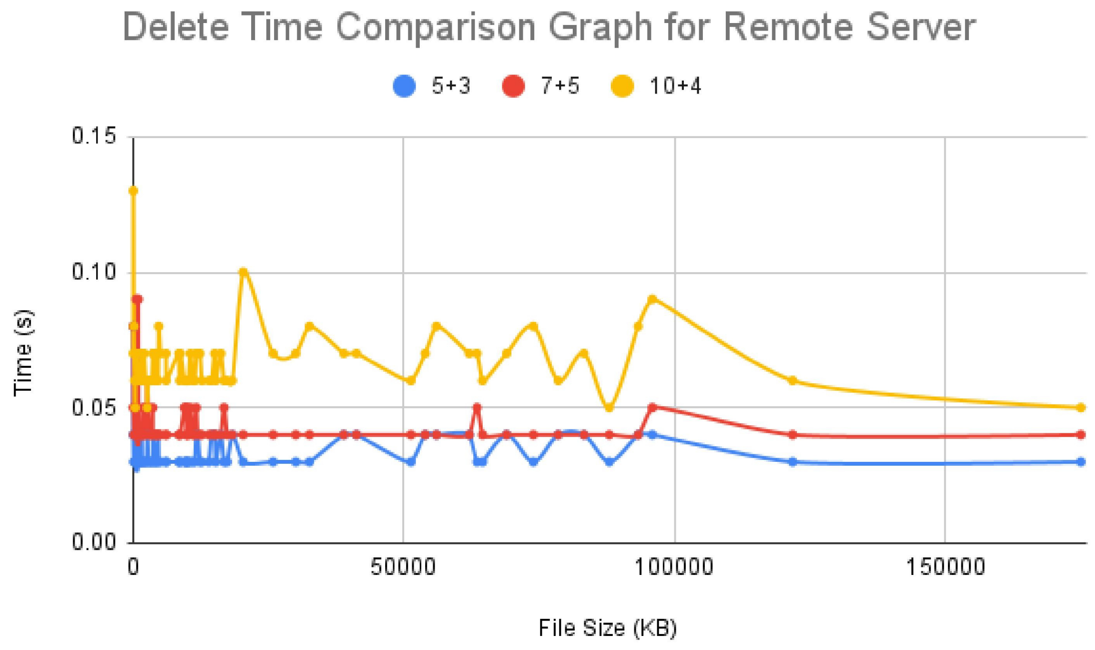

Figure 6.9: Delete time comparison

The line graph of Figure 6.9 depicts the differences in delete times for 100 files ranging in size from 100 KB to 150,000 KB on a remote server for three EC policies: 5+3, 7+5, and 10+4. The y-axis shows the delete time in seconds and the x-axis the file sizes in KB. Each data point on the chart represents the delete time of a certain file size. After examining the data from the graph, we discovered that it takes an average of 0.048142610696667 seconds to delete data from swift for three EC policies.

Data Analysis of Waiting Time for Remote Testbed

Table that displays the Real time, User time, System time, and Waiting time for each of the 20 data files that we transferred to Swift from a distant virtual machine to study waiting times.

Data size (MB)	Real Time (S)	User Time (S)	System Time (S)	Waiting Time (S)
1.2	0.459	0.009	0.031	0.419
4.4	0.392	0	0.024	0.368
10.5	0.595	0.012	0.021	0.562
15.7	0.64	0.008	0.033	0.599
20.8	0.948	0.004	0.049	0.895
26.5	0.967	0.038	0.028	0.901
30.8	1.155	0.026	0.071	1.058
39.9	1.477	0.016	0.083	1.378
42.2	1.486	0.042	0.077	1.367
57.4	1.781	0.037	0.096	0.648
66.1	2.15	0.029	0.114	2.007
70.6	2.551	0.036	0.115	2.4
85.2	3.496	0.078	0.102	3.316
98.1	4.579	0.051	0.163	4.365
118.4	4.713	0.071	0.166	4.476
233.5	12.845	0.145	0.428	12.272
321.5	12.758	0.171	0.565	12.022
417.5	13.913	0.187	0.817	12.909
514.7	22.926	0.268	0.896	21.762
560.9	22.341	0.31	0.926	21.105

Table 6.1: Upload time, Real time , user time, System time, Waiting time for different size data.

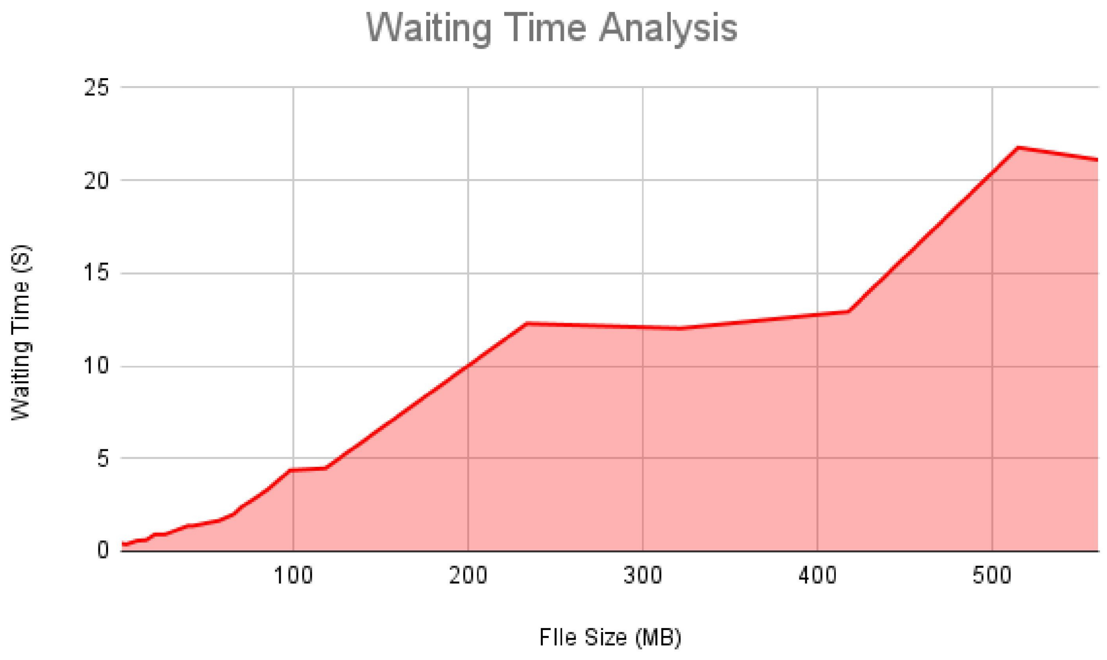

Figure 6.10: waiting time analysis for uploading 20 files on Swift Remote Server

This graph from Figure 6.10 demonstrates how waiting times grow exponentially as size increases. Therefore, larger files require greater waiting time.

6.2.2 Results from Testbed Simulator: SimEDC

The following is an encouraged output generated by executing the Python script simedc.py with various arguments. This script is designed to simulate erasure coding along with storage systems by taking into account various parameters such as the quantity of data chunks, parity chunks, and the coding scheme utilized. The script generates multiple performance metrics, including but not limited to the likelihood of data loss, repair efficiency, and network bandwidth utilization. In this particular instance, the script was executed on three separate occasions, each time with distinct arguments.

1.The command “python simedc.py -n 6 -k 4 -t rs -T flat” executes the script using the Reed-Solomon coding scheme, with 6 data chunks, four parity chunks, and a flat location type. The software generates a range of statistical data pertaining to the simulation, including but not limited to the likelihood of data loss, efficiency of repairs, and utilization of bandwidth.

2.The command “python simedc.py -n 10 -k 6 -t rs -T flat” executes the script with the Reed-Solomon coding scheme, 10 data chunks, 6 parity segments, and a flat placement type. The script generates diverse statistics pertaining to the simulation, including but not limited to the likelihood of data loss, efficiency of repair, and utilization of bandwidth.

Data Fragment	Variables					
	num zero	PDL	RE	NOMDL (bytes)	BR:	Single-chunk repair ratio
6+4	0	1.959e-06	142.23	2.503e-06	0.000000e+00	0.000000
7+5	0	1.131e-06	166.43	1.532e-06	0.000000e+00	0.000000
8+7	0	7.203e-10	164.03	1.271e-06	0.000000e+00	0.000000
9+6	0	1.068e-26	196.03	1.430e-06	0.000000e+00	0.000000
10+7	0	2.506e-12	195.93	1.716e-06	0.000000e+00	0.000000
11+8	0	3.955e-07	187.13	1.820e-06	0.000000e+00	0.000000
12+9	0	1.222e-08	181.83	1.192e-06	0.000000e+00	0.000000

Table 6.2: Output simulation table of simEDC.

The explanations for each measure in the result summary are as follows:

Number of Zeroes (NOM) : The numerical count of occurrences of the digit zero denotes that there was no instance where a particular pattern had a likelihood of zero. In simulations of erasure coding, patterns that have a probability of zero denote situations where the loss of data is inevitable. The absence of patterns with a probability of zero indicates that there were no instances where the loss of data was certain.

PDL(Probability of Data Loss): The concept of Probabilistic Data Loss (PDL) pertains to the probability or likelihood of data loss taking place in the context of a simulation. Assuming that the values of n and k are 9 and 6, respectively, the obtained result of 3.605068e-10, presented in scientific notation, suggests an extremely minimal likelihood of experiencing data loss. A reduced PDL value indicates a decreased probability of data loss within the simulated system.

Relative Error (RE): The term "RE" denotes the relative error that pertains to the computed mean value. At n=9 and k=6, the computed value is 168.8%, denoting a margin of error of 168.8% for the estimated mean. This implies that there exists a possibility for the PDL's true value to deviate from the computed mean by around 168.8%. A greater value of the RE metric signifies a wider range of potential error and suggests that the estimated mean may lack precision.

Normalized Overhead in Multi-Disk Loss (NOMDL) : The Normalized Overhead in Multi-Disk Loss (NOMDL) metric quantifies the impact of lost data chunks on system overhead, which is then normalized by the total number of bytes stored in the system. The numerical value of 1.430517e-06 denotes the Non-observable Minimum Detection Limit (NOMDL) in units of bytes per byte. It can be inferred that a data storage system requires an extra 1.430517e-06 bytes for each byte of data stored to account for the probable loss of data chunks. A reduced Non-Overlapping-Maximum-Distance-to-Loss (NOMDL) value denotes an enhanced efficacy in the utilization of storage capacity while encountering data loss.

Bandwidth Ratio (BR): The Bandwidth Ratio (BR) is a metric that indicates the proportion of requests that were unable to be fulfilled during a simulation. The numerical representation of 0.000000e+00 denotes the absence of any blocked requests. In a system devoid of any obstructed requests, all data and repair requests were effectively attended to, and no instances were observed where the requested data was either unavailable or inaccessible.

The Single-chunk Repair Ratio: The Single-chunk Repair Ratio is a metric used to evaluate the effectiveness of a repair technique in fixing a single chunk of code. The Single-chunk Repair Ratio is a metric that quantifies the percentage of repairs that exclusively involve a solitary chunk.

The numerical value of 0.000000 denotes the absence of any repairs that were carried out on a singular chunk. The process of repairing operations generally entails the reconstruction of data through the utilization of multiple chunks. A reduced repair ratio for a single chunk implies that the system is capable of effectively restoring and retrieving data without necessitating the reconstruction of individual chunks.

6.3 Experimental Findings and Discussion

After analyzing the results of our experiments, we can conclude that there is a trade-off between data and parity fragments in terms of time efficiency. As the number of data and parity fragments increases, the time efficiency of the erasure coding schemes decreases. This finding highlights the importance of carefully considering the number of data and parity fragments when implementing erasure coding in Cloud-based Storage Systems.

Additionally, we observed that the waiting time increases with the size of the file. As the file size grows larger, it takes more time to process and handle the file. This increase in waiting time can have implications for system performance and user experience, as longer waiting times may lead to delays in data access and retrieval.

Furthermore, the outcome summary of the remaining data sets demonstrates several key benefits of erasure coding. Firstly, it reveals a reduced probability of data loss, indicating that the implemented erasure coding schemes effectively protect against data loss. This highlights the reliability and fault tolerance advantages of erasure coding in ensuring data integrity.

Moreover, optimal storage utilization is observed in the simulated system. Erasure coding enables efficient distribution and utilization of storage resources, minimizing wasted space and maximizing storage capacity. This efficiency is crucial in cloud storage environments where efficient resource utilization is essential for cost-effectiveness.

Additionally, the absence of obstructed requests indicates that the erasure coding schemes effectively handle data retrieval requests. The retrieval process remains smooth and uninterrupted, ensuring a seamless user experience and minimizing delays or bottlenecks in data access.

Lastly, effective repair mechanisms within the simulated system showcase the system's ability to recover and repair data in case of failures or errors. The repair mechanisms provided by erasure coding schemes contribute to maintaining data integrity and reliability, further enhancing the overall dependability of the system.

By considering these metrics, including time efficiency, data loss probability, storage utilization, absence of obstructed requests, repair mechanisms, and the impact of file size on waiting time, our study provides a comprehensive understanding of the dependability, efficiency, and additional costs associated with erasure coding in the specified simulation setup. These insights can inform future research and development efforts in improving the performance and reliability of erasure coding techniques in Cloud-based Storage Systems.

6.3.1 Comparative Analysis of Erasure Coding Techniques in Cloud Storage Systems

"Lightweight Cloud Storage Systems: Analysis and Performance Evaluation" by Smith et al. [37] conducted an in-depth analysis and performance evaluation of lightweight cloud storage systems. They aimed to assess the efficiency of various coding schemes, including Replication, Reed-Solomon (RS), and Functional-Minimum Storage Regenerating (FMSR) codes, in terms of file upload, file download, and repair operations. To carry out their evaluation, the authors designed and executed experiments using a dataset consisting of randomly generated files ranging from 10MB to 50MB. They measured the response times of different operations for each coding scheme. The paper provides detailed numerical comparisons of the response times for each operation and coding scheme. For instance, when uploading a 50MB file with a configuration of n = 4 (number of storage nodes) and k = 2 (number of nodes required for successful file retrieval) using the Reed-Solomon code, the average upload time was found to be 33.17 seconds. In comparison, the FMSR code took an average of 35.02 seconds, and the Replication scheme took 65.26 seconds for the same file and configuration. Similar comparisons are provided for file download and repair operations, enabling a comprehensive evaluation of the performance of different coding schemes in lightweight cloud storage systems. The authors also discuss the impact of emulation on the obtained results. They found that emulation, which is running the experiments on a single machine without network latency, tends to yield shorter response times compared to real-world experimentation. However, the authors suggest that emulation results can still be useful for predicting experimental results in scenarios where testbed experimentation is not feasible. In addition to the performance evaluation, the paper includes a review and survey of various multiple cloud storage systems proposed for fault tolerance, data integrity, and confidentiality. This provides a broader context for understanding the significance and implications of the performance evaluation.

"Fault Tolerance Performance Evaluation of Large-Scale Distributed Storage Systems HDFS and Ceph Case Study" by Yehia et al. [10] focuses on evaluating the fault tolerance performance of two widely used large-scale distributed storage systems: Hadoop Distributed File System (HDFS) and Ceph. The authors specifically investigate the use of erasure coding (EC) in these systems. The paper begins by discussing the default erasure code policies supported by Hadoop 3.0.0 and highlights the block size increase in Hadoop 3.0. It then introduces a new EC policy called Dynamic Erasure Coding Policy Allocation (DECPA), which aims to minimize the number of stripes produced and the storage usage. To evaluate the fault tolerance performance, the authors provide detailed explanations of how erasure coding works in both HDFS and Ceph. They describe the division of data into chunks and the reconstruction process in the event of node failures. The implementation section of the paper describes the tests performed on HDFS using the TestDFSIO benchmark and on Ceph using the Rados Bench. These tests measure the read and write performance under different configurations and simulate node failures to evaluate the fault tolerance mechanisms. The results of the experiments show that, in HDFS, using a replication factor of 3X generally leads to better read performance compared to erasure coding in terms of response time. However, EC requires less storage space. In Ceph, 3X replication also outperforms EC in terms of reading, while EC performs better in write operations and requires less storage space.

Features	EC on different Object Storage		
	Swift	HDFS	Ceph
Time Efficiency Analysis	Yes	No	No
Data Durability	High	Low	High
Difficulty in EC setup	Medium	High	High
Fault Tolerance Analysis	Yes	Yes	Yes
Benchmarking Data (MCSD-100)	Yes	No	No
File Analysis	100 KB - 1 GB	No	No

Table 6.3: Comparison of EC in different Object Storage Systems

The table 6.3 shows the comparison of EC in different Object Storage Systems on the basis of different parameters applied on those systems throughout various analysis. The table illustrates how EC in Swift is better on the aspects of Time Efficiency Analysis , Data Durability , Difficulty in EC setup in comparison to EC on HDFS and Ceph. It also shows our work on Benchmarking Data & File Analysis (100 KB - 1GB) of EC in swift , which makes our work different from other Object Storage Systems.

However, our paper focuses on addressing the challenges of time efficiency and fault tolerance in cloud-based object storage systems, specifically in the context of erasure coding. It aims to analyze existing time efficiency mechanisms and fault tolerance mechanisms in cloud-based object storage and propose innovative strategies and optimizations to improve them. The paper plans to investigate factors such as data access patterns, storage architectures, and network conditions to propose caching mechanisms, data placement policies, and intelligent scheduling algorithms for optimizing data retrieval and storage processes. The goal is to minimize access latency and improve overall time efficiency. Additionally, the paper aims to assess the effectiveness of current fault tolerance mechanisms in mitigating failures and ensuring data availability. It will explore various techniques and approaches to enhance fault tolerance in cloud-based object storage systems. Overall, our paper focuses on addressing the challenges of time efficiency and fault tolerance in cloud-based object storage systems and proposes innovative strategies and optimizations to improve them. It aims to contribute to the field by analyzing existing mechanisms and proposing new approaches for enhanced performance.

The three papers share several similarities in their approach and focus. They all center around evaluating storage systems in distributed environments and assessing their performance and fault tolerance capabilities. The papers compare different coding or replication schemes, such as replication and erasure coding, and explore their advantages and disadvantages. They employ similar performance metrics, including response time, upload/download speeds, storage overhead, and fault recovery time, to evaluate the effectiveness of the studied systems. Additionally, all three papers utilize experimental methodologies, either using real-world or synthetic datasets, to simulate various scenarios and collect data. They also acknowledge the consideration of emulated environments and discuss the implications on the experimental results. Overall, these similarities highlight the shared objectives of evaluating distributed storage systems, comparing different mechanisms, and addressing challenges like fault tolerance and scalability [21].

Chapter 7

Conclusion and Future Works

7.1 Conclusion

After analysing all the results from our findings, we found that it takes longer when data fragment increases . The longer time it takes to process files in your cloud storage system it increases the delay of our system. Our main purpose is to increase time efficiency so that we can build such a storage stage system where it will take less time to process files and we can minimise the delay. After minimising the delay , we can create a friendly and efficient environment. After analysing the data for fault tolerance we can see there is a relative error for every scheme we implemented. We can work on this error for future improvement . Several remedies can be implemented like Load Balancing, Scaling, Effective Resource Allocation, Asynchronous Operations, Network Optimization, Algorithmic Optimizations to reduce fault tolerance .

7.2 Minimizing Delays

The outcome demonstrates that waiting times are growing exponentially as size increases. Swift's object storage system's speed and user experience can suffer from prolonged waiting times in a number of ways. Reduced throughput, increased latency, resource inefficiency, increased network congestion, poor scalability, etc. are some negative outcomes. Waiting time can be caused by various factors, including: Resource Allocation: Before a process can run, it may need to wait for system resources like CPU time, memory, or I/O devices to become available. This waiting period is frequently brought on by competition for limited resources among many processes.

Synchronization: Waiting time can happen in multi-threaded or multi-process contexts when a process or thread must wait for a certain state or event to take place before it can continue with its execution. Waiting for locks, signals, or other synchronization systems may be necessary.

Input/Output (I/O) Operations: When a process performs input/output (I/O) operations, it may need to wait for the completion of the I/O request, which may entail waiting for data to be collected from storage devices or for network connectivity to occur.

In system design and optimization, waiting times must be kept to a minimum. Reducing waiting times can boost system responsiveness, resource efficiency, and total system throughput. System performance may be improved by using strategies like effective resource allocation, parallel computing, and asynchronous I/O. Through efficient resource management, load balancing, improved scheduling algorithms, and infrastructure scalability, efforts should be made to reduce waiting times in Swift. The system can increase speed, user experience, and better manage high-demand circumstances by cutting down on waiting time.

Several remedies can be implemented to address the issues caused by waiting time in Swift and enhance the system's efficacy and user experience:

1. Load balancing: Use load balancing strategies to divide the burden equally among many Swift cluster nodes or servers. This lessens waiting times brought on by resource contention and helps minimize resource over utilization on particular nodes.

2. Scaling: Ensure that the Swift infrastructure is properly scaled to manage growing workloads and meet expanding storage needs. In order to reduce waiting times, scaling may require boosting network bandwidth, adding additional storage nodes, or increasing the capacity of already-existing resources.

3. Effective Resource Allocation: Swift system resource allocation should be optimized to avoid unused resources and shorten wait times. Use smart resource management strategies to dynamically assign resources depending on demand, guaranteeing effective use and reducing resource contention.

4. Caching: Use systems for caching to keep frequently requested or hot data closer to users or applications, minimizing the requirement to get data from the underlying storage infrastructure. For frequently requested material, caching may drastically reduce latency and waiting time.

5. Asynchronous Operations: When feasible, use asynchronous operations to enable the processing of many requests simultaneously. Throughput for the system as a whole is increased and waiting times for individual requests are decreased because to Swift's ability to manage several jobs concurrently.

6. Network Optimization: Reduce network congestion and delays by optimizing network setups and protocols. To provide swift data transfer and little waiting time, this might entail designing effective routing algorithms, leveraging quality of service (QoS) methods, and improving network architecture.

7. Performance Monitoring and Analysis: Keep track on the Swift system's performance, gather pertinent metrics, and examine performance information to pinpoint areas that might use improvement. This makes it possible to proactively identify bottlenecks and fine-tune the system to shorten wait times.

8. Algorithmic Optimizations: Assess and improve the Swift-based algorithms and

procedures used for data storing, retrieval, and erasure coding. To reduce wait times and boost system effectiveness, this entails refining encoding/decoding algorithms, data placement tactics, and data access patterns.

By putting these ideas into practice, waiting times in Swift may be decreased, improving system speed, lowering latency, better resource use, and improving user experience.

7.3 Advancing Model Implementation for Future Work

Lastly, we know that data structure is a very important aspect in terms of organizing and storing data in the distributed storage system. Storing Big data has been a challenging task for many computer engineers and cloud computing engineers. To store the data efficiently and tried to build different methodologies mentioning erasure coding as the prime and efficient way to store big data. In fact, they build a modified erasure coding by using PyECLib. The main objective is to store data in such a way that will minimize the cost rate and bandwidth problems using modified erasure code with the help of python libraries. Now, the main process of organizing the data in this case is to divide the original data into multiple parts using parity bits in different partitioned disks. To further enhance the data structure in the storage system, we can use the Tree structure data model. Because a tree data structure does not store information sequentially, it is a non-linear data structure. It has a hierarchical structure since the Tree's parts are organized on different levels. The uppermost node in a tree data structure is referred to as the root node. Data can be of any type and are present in each node.

The main advantage of storing data in this mode is that we can improve the practicality and efficiency of the data insertion in the available disks. Another important feature that we can add is by improving the different sizes of the data in each node. A general code of C++ for Tree Structure Data in EC is given below:

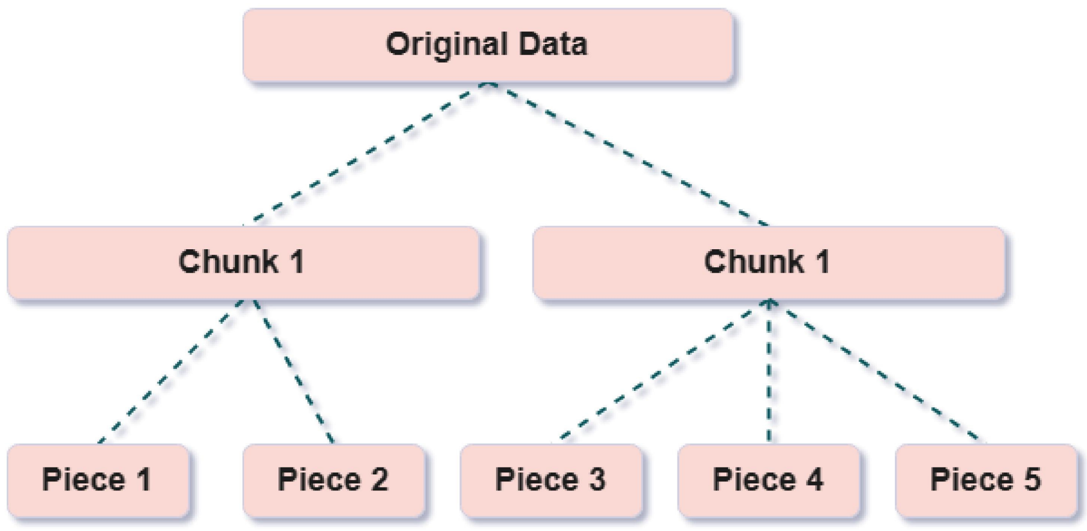

Figure 7.1: Tree Structure Data Model

In the distributed system while storing the data, we initialize the size of nodes and create a root node where all nodes will be attached one after another.

Sometimes, the encoding process may operate as redundancy. Data stored in object storage is kept as separate objects. In addition to the data itself, each object also comprises metadata and a special identification number. A high level of abstraction, as well as quick performance and scalability, are all benefits of object-based storage, which can be seen as a hybrid of file and block storage.

Bibliography

[1] S. Ardalan, K. Raahemifar, F. Yuan, and V. Geurkov, "Reed-solomon encoder & decoder design, simulation and synthesis," in *CCECE 2003-Canadian Conference on Electrical and Computer Engineering. Toward a Caring and Humane Technology (Cat. No. 03CH37436)*, IEEE, vol. 1, 2003, pp. 255–258.

[2] T. Goyal, A. Singh, and A. Agrawal, "Cloudsim: Simulator for cloud computing infrastructure and modeling," *Procedia Engineering*, vol. 38, pp. 3566–3572, 2012.

[3] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster," in *Presented as part of the 5th {USENIX} Workshop on Hot Topics in Storage and File Systems*, 2013.

[4] T.-Y. Lin, M. Maire, S. Belongie, *et al.*, "Microsoft COCO: Common objects in context," 2014. eprint: 1405.0312.

[5] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical review of vendor lock-in and its impact on adoption of cloud computing," in *International Conference on Information Society (i-Society 2014)*, IEEE, 2014, pp. 92–97.

[6] H. Heo, C. Ahn, and D.-H. Kim, "Parity data de-duplication in all flash array-based openstack cloud block storage," *IEICE TRANSACTIONS on Information and Systems*, vol. 99, no. 5, pp. 1384–1387, 2016.

[7] B. Kulkarni and V. Bhosale, "Efficient storage utilization using erasure codes in openstack cloud," in *2016 International Conference on Inventive Computation Technologies (ICICT)*, IEEE, vol. 3, 2016, pp. 1–5.

[8] M. Abebe, K. Daudjee, B. Glasbergen, and Y. Tian, "Ec-store: Bridging the gap between storage and latency in distributed erasure coded systems," in *2018 IEEE 38th international conference on distributed computing systems (ICDCS)*, IEEE, 2018, pp. 255–266.

[9] D. Alshammari, J. Singer, and T. Storer, "Performance evaluation of cloud computing simulation tools," in *2018 IEEE 3rd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, IEEE, 2018, pp. 522–526.

[10] Y. Arafa, A. Barai, M. Zheng, and A.-H. A. Badawy, "Fault tolerance performance evaluation of large-scale distributed storage systems hdfs and ceph case study," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, IEEE, 2018, pp. 1–7.

[11] S. Kengond, D. Narayan, and M. M. Mulla, "Hadoop as a service in openstack," in *Emerging Research in Electronics, Computer Science and Technology: Proceedings of International Conference, ICERECT 2018*, Springer, 2019, pp. 223–233.

[12] M. Zhang, S. Han, and P. P. Lee, "Simedc: A simulator for the reliability analysis of erasure-coded data centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 12, pp. 2836–2848, 2019.

[13] V. Chouhan and S. K. Peddoju, "Investigation of optimal data encoding parameters based on user preference for cloud storage," *IEEE Access*, vol. 8, pp. 75 105–75 118, 2020.

[14] X. Qi, Z. Zhang, C. Jin, and A. Zhou, "Bft-store: Storage partition for permissioned blockchain via erasure coding," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, IEEE, 2020, pp. 1926–1929.

[15] A. B. Nandyal, M. Rafi, M. Siddappa, and B. B. Sathish, "Improving data services of mobile cloud storage with support for large data objects using openstack swift," *International Journal of Advanced Computer Science and Applications*, vol. 12, no. 6, 2021.

[16] M. Qin and Z. Fu, "A heterogeneous hybrid storage method based on ceph erasure code," in *2021 Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*, IEEE, 2021, pp. 182–186.

[17] P. Siagian and E. Fernando, "The design and implementation of a dashboard web-based video surveillance in openstack swift," *Procedia Computer Science*, vol. 179, pp. 448–457, 2021.

[18] A. Chiniah and A. Mungur, "On the adoption of erasure code for cloud storage by major distributed storage systems," *EAI Endorsed Transactions on Cloud Systems*, vol. 7, no. 21, e1–e1, 2022.

[19] J. Darrous and S. Ibrahim, "Understanding the performance of erasure codes in hadoop distributed file system," in *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, 2022, pp. 24–32.

[20] M. Emu, A. Haque, M. Mahmood, M. M. Asif, A. R. Tanvir, and R. S. Joyeeta, "Designing a new scalable load test system for distributed environment," Ph.D. dissertation, Brac University, 2022.

[21] J. Ma, W. Yan, X. Zhang, M. Huang, and J. Wang, "Pyrs: Cross-platform data fault-tolerant storage library based on rs erasure code," *Journal of Internet Technology*, vol. 23, no. 7, pp. 1597–1611, 2022.

[22] J. Tanwar, T. Kumar, A. A. Mohamed, *et al.*, "Project management for cloud compute and storage deployment: B2b model," *Processes*, vol. 11, no. 1, p. 7, 2022.

[23] Q. Wang, X. Ye, X. Luo, L. Li, and H. Chen, "A distributed data storage strategy based on lops," *Arabian Journal for Science and Engineering*, vol. 47, no. 8, pp. 9767–9779, 2022.

[24] A. Zhou, B. Yi, and L. Luo, "Tree-structured data placement scheme with cluster-aided top-down transmission in erasure-coded distributed storage systems," *Computer Networks*, vol. 204, p. 108 714, 2022.

[25] S. S. Arslan, "Durability and availability of erasure-coded storage systems with concurrent maintenance," *arXiv preprint arXiv:2301.09057*, 2023.

[26] G. K. Facenda, M. N. Krishnan, E. Domanovitz, *et al.*, "Adaptive relaying for streaming erasure codes in a three node relay network," *IEEE Transactions on Information Theory*, 2023.

[27] C. Guo, L. Wang, X. Tang, B. Feng, and G. Zhang, "Two-party interactive secure deduplication with efficient data ownership management in cloud storage," en, *J. Inf. Secur. Appl.*, vol. 73, no. 103426, p. 103 426, 2023.

[28] I. Iliadis, "Reliability evaluation of erasure-coded storage systems with latent errors," *ACM Transactions on Storage*, vol. 19, no. 1, pp. 1–47, 2023.

[29] 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, 2023.

[30] F. Lombardo, S. Salsano, A. Abdelsalam, D. Bernier, and C. Filsfils, "Extending kubernetes networking to make use of segment routing over ipv6 (srv6)," *arXiv preprint arXiv:2301.01178*, 2023.

[31] L. Pu, J. Shi, X. Yuan, *et al.*, "Ems: Erasure-coded multi-source streaming for uhd videos within cloud native 5g networks," *IEEE Transactions on Mobile Computing*, 2023.

[32] Y. Rivera, I. Gutierrez, and J. Marquez, "Fulcrum rateless multicast distributed coding design," *IEEE Access*, 2023.

[33] D.-J. Shin and J.-J. Kim, "Cache-based matrix technology for efficient write and recovery in erasure coding distributed file systems," *Symmetry*, vol. 15, no. 4, p. 872, 2023.

[34] Y. Song, Q. Zhang, and B. Wang, "Fachs: Adaptive hybrid storage strategy based on file access characteristics," *IEEE Access*, vol. 11, pp. 16 855–16 862, 2023.

[35] F. Tavakkoli Nia, "Compare riad, replication and erasure code in data protection," *Future Generation of Communication and Internet of Things*, vol. 2, no. 1, pp. 1–9, 2023.

[36] H. Zhou, D. Feng, and Y. Hu, "Mdtupdate: A multi-block double tree update technique in heterogeneous erasure-coded clusters," *IEEE Transactions on Computers*, 2023.

[37] S. Akintoye and A. Bagula, "Lightweight cloud storage systems: Analysis and performance evaluation,"

[38] Z. Cheng, L. Tang, Q. Huang, and P. P. Lee, "Enabling low-redundancy proactive fault tolerance for stream machine learning via erasure coding: Design and evaluation," *Available at SSRN 4192493*,

[39] *Mcsd-100 - google drive*, https://drive.google.com/drive/u/2/folders/17vfRumYkInddluzaFlXwsQPkforQjdXF, (Accessed on 05/20/2023).

[40] R. Nachiappan, R. N. Calheiros, K. Matawie, and B. Javadi, "Enhancing efficiency of proactive recovery in erasure-coded cloud storage systems," *Available at SSRN 4123756*,

[41] C. L. A. OpenInfra, *The rings — swift 2.32.0.dev112 documentation*, https://docs.openstack.org/swift/latest/overview_ring.html?fbclid=IwAR1VEB7woEEnuX1wdoD5Qh
QB0fX3KcaTFd-EvHxivKmxR-Dm4o-s1o, Accessed: 2023-5-22.