A Novel Cloud Storage Ecosystem for Efficient and Secured Multimedia Services

by

Jannatun Noor Mukta

DOCTOR OF PHILOSOPHY

Department of Computer Science & Engineering

BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

(BUET)

Dhaka 1000

June, 2023

PhD Thesis

A Novel Cloud Storage Ecosystem for Efficient and Secured Multimedia Services

A thesis submitted to the

Department of Computer Science & Engineering

In partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY
IN
COMPUTER SCIENCE AND ENGINEERING

by

Jannatun Noor Mukta (0419054004 P)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
DHAKA 1000, BANGLADESH

June, 2023

Dedicated to my loving parents

Author's Contact

Jannatun Noor Mukta

Candidate for degree of Doctor of Philosophy

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology (BUET), Dhaka.

Email: `jannatun.noor@bracu.ac.bd`

CANDIDATES' DECLARATION

This is to certify that the work presented in this thesis, titled, 'A Novel Cloud Storage Ecosystem for Efficient and Secured Multimedia Services', is the outcome of the investigation and research carried out by me under the supervision of Dr. A. B. M. Alim Al Islam, in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka - 1000.

It is also declared that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Noor

Jannatun Noor Mukta

Candidate

The thesis titled "A NOVEL CLOUD STORAGE ECOSYSTEM FOR EFFICIENT AND SECURED MULTIMEDIA SERVICES" submitted by Jannatun Noor Mukta, Roll No. 0419054004, Session APRIL-2019, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and Engineering and approved as to its style and contents on June 04, 2023.

Board of Examiners

1. [illegible] Dr. A. B. M. Alim Al Islam Professor Department of CSE, BUET, Dhaka	Chairman (Supervisor)
2. [illegible] Dr. Mahmuda Naznin Head and Professor Department of CSE, BUET, Dhaka	Member (Ex-Officio)
3. [illegible] Dr. Abu Sayed Md. Latiful Hoque Professor Department of CSE, BUET, Dhaka	Member
4. [illegible] Dr. Md. Mostofa Akbar Professor Department of CSE, BUET, Dhaka	Member
5. [illegible] Dr. Muhammad Abdullah Adnan Professor Department of CSE, BUET, Dhaka	Member
6. [illegible] Dr. Rezwana Reaz Assistant Professor Department of CSE, BUET, Dhaka	Member
7. [illegible] Dr. Md. Abdur Razzaque Professor Department of CSE, University of Dhaka, Dhaka	Member
8. [illegible] Dr. M. Kaykobad Distinguished Professor Department of CSE, BRAC University, Dhaka	Member (External)

Acknowledgement

Foremost, my utmost gratitude to the Almighty Allah Subhanahu Tayala for blessing me with the opportunity, strength, and resources to work on and complete this research work. Then, I express my heart-felt gratitude to my supervisor, Dr. A. B. M. Alim Al Islam, for his constant supervision of this work. He helped me a lot in every aspect of this work and guided me with proper directions whenever I sought one. His patient hearing of my ideas, critical analysis of my observations, and detecting flaws (and amending thereby) in my thinking and writing have made this thesis a success.

I would also like to thank the members of my thesis committee: Prof Dr. Mahmuda Naznin, Prof. Dr. Abu Sayed Md. Latiful Hoque, Prof. Dr. Md. Mostafa Akbar, Prof. Dr. Mohammand Abdullah Adnan, Dr. Rezwana Reaz, Prof. Dr. Md. Abdur Razzaque, and specially the external member Prof. Dr. M. Kaykobad, for their encouragement, insightful comments, and valuable suggestions.

I am also thankful to Prof. Dr. Md. Shohrab Hossain (Professor, CSE, BUET), Dr. Mahbub Majumder (Dean, SDS, BRAC University), Dr. Sadia Hamid Kazi (Chairperson, CSE, BRAC University), and my colleagues from BRAC University who always help me whenever I seek help and need official leave to complete the coursework and thesis work. Additionally, I want to express my heartfelt gratitude to my sincere, talented, hard working, and dedicated Research Assistants Joyanta Jyoti Mondal and Nazrul Huda Shanto. Besides, I like to thank Sadiqul Sakif, Rezwanul Ratul, Mir Rownak, my thesis students, Dr. Novia Nurain, and Ishrat Jahan for their help and valuable suggestions regarding writing and presentation of this thesis.

I believe, only I did not work for the Ph.D, rather my whole family did the same. I am grateful to my little son Aanas Noor (6 y), my daughter Ayisha Noor (only 1.5 y), my better half Dr. Md. Golam Zel Asmaul Husna, and my mother Jahanara Rahman. Their sacrifice and continuous support helped me a lot every time. Without their cooperation, it would never be possible to finish my work. Throughout the Ph.D period, many times I decided to quit my Ph.D. Playing a diversified role, as a mother of two children, as a faculty, as a thesis supervisor, and as a Ph.D. student, I sometimes feel lost and can not breathe. That time, two people always encouraged me to continue my research work - my supervisor and my better half. I do not know how to express my gratitude towards them.

Moreover, I extend my appreciation to the Deputyship for Research & Innovation, Ministry of Education in Saudi Arabia for funding this research work through project 764 number (DRI-KSU-762). Special thanks to Prof. Dr. Najla Abdulrahman Al-Nabhan for her kind collaboration. Last but not the least, I remain ever grateful to my beloved parents, who always exist as sources of inspiration behind every success of mine.

Abstract

Since massive numbers of images are now being communicated from, and stored in different cloud systems, faster retrieval has become extremely important. This is more relevant, especially after COVID-19 in bandwidth-constrained environments. However, to the best of our knowledge, a coherent solution to overcome this problem is yet to be investigated in the literature. Hence, by customizing the Progressive JPEG method, we propose a new Scan Script and a new lossy PJPEG architecture to reduce the file size as a solution to overcome our Scan Script's drawback. We improve the scanning of Progressive JPEG's picture payloads to ensure Faster Image Retrieval using the change in bit pixels of distinct Luma and Chroma components (Y, C_b, and C_r). The orchestration improves user experience even in bandwidth-constrained cases. We evaluate our proposed orchestration in a real-world setting across two continents encompassing a private cloud. Compared to existing alternatives, our proposed orchestration can improve user waiting time by up to 54% and decrease image size by up to 27%. Our proposed work is tested in cutting-edge cloud apps, ensuring up to 69% quicker loading time.

In addition, the demand for a robust as well as a highly-available surveillance system with efficient media sharing capability has considerably risen in recent times. To face these challenges, we propose a new methodology to utilize OpenStack Swift's object storage to efficiently store and archive media data. Our method leverages expanding the cloud file sharing capabilities from storing media files to also processing and archiving them along with performing encryption. Our proposed approach first segments, encodes, and transcodes the videos according to several resolutions for covering diversified remote devices. Then, we store the processed video footage in the storage server of OpenStack Swift. Afterwards, we perform necessary media encryption-decryption, compress the files containing the video data, and archive them using an archive server. We carry out rigorous experiments over a real setup comprising machines deployed in two different countries (Canada and Bangladesh), located over two different continents, to validate the efficacy and efficiency of our proposed architecture and methodology. Experimental results demonstrate substantial performance improvement using our approach over conventional alternative solutions.

Moreover, we propose a novel Content-Based Searching (CoBS) architecture to extract additional information from images and documents and store it in an Elasticsearch-enabled database, which helps us to search for our desired data based on its contents. This approach works in two sequential stages. First, it will be uploaded to a classifier that will select the data type and send it to the specific model for the data. The images that are being uploaded are sent to our trained model for object detection, and the documents are sent for keyword extraction. Next, the extracted information is sent to Elasticsearch, which enables searching based on the contents. We train our models with comprehensive datasets (Microsoft COCO Dataset) for multimedia data and SemEval2017 Dataset for document data. Besides, we propose a generalized architecture for smooth and efficient management as well as retrieval of multimedia data in cloud systems. Here, we demonstrate that video segment download time improves by up to 30% when segmentation is done in the object server rather than in the proxy server. After, we present a generalized architecture named *'RemOrphan'* for detecting the orphan garbage data using OpenStack Swift hash Ring and scripts. We deploy a private media cloud SPMS and find that around 35% data can be orphan garbage data. Due to the huge amount of orphan data, rsync replication needs higher time and more network overhead which hampers the system sustainability. We lower around 25% sync delay and 30% network overhead after deploying a deletion daemon to remove the orphan garbage data.

Furthermore, we propose a test-driven automated architecture for load testing, named as 'svLoad' to compare the performance of cache and backend servers. Here, we designed test cases considering diversified real scenarios such as different protocol types, same or different URLs, with or without load, cache hit or miss, etc. using tools namely JMeter, Ansible, and some custom utility bash scripts. To validate the efficacy of our proposed methodology, we conduct a set of experiments by running these test cases over a real private cloud development setup using two open source projects - Varnish as the cache server and OpenStack Swift as the backend server. Our focus is also to find out bottlenecks of Varnish and Swift by executing load requests, and then tune the system based on our load test analysis. After successfully tuning the Swift, Varnish, and network system, based on our test analysis, we were able to improve the response time by up to 80%.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A cloud storage ecosystem [17, 18] is a complex system of interdependent components that all work together to enable efficient and secured cloud storage services having retrieval, searching, archiving, faster inter-communication from one server to another server, etc (in Figure 1.1). Besides, with the rapid growth of embedded devices, media industries have started facing challenges in storing, processing, and managing large amounts of multimedia data including video, photos, audio, and text through several Software-as-a-Service applications [3]. All these applications present a common

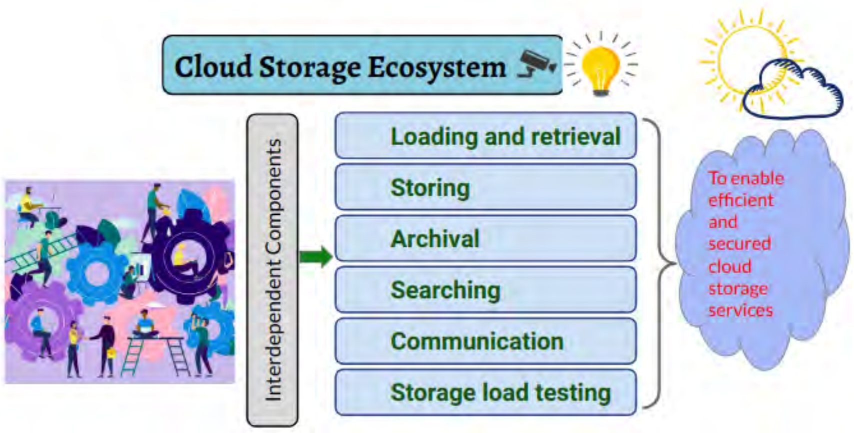


Figure 1.1: Interdependent components of a cloud storage ecosystem

need for an easily-accessible and sustainable cloud storage ecosystem for multimedia data that can potentially grow without bounds or limits (in Figure 1.2 and Figure 1.3). For building a novel storage ecosystem to embrace this demand, some major components under consideration are - a) data loading and retrieval, b) efficient multimedia data storing, c) archival, d) searching, e) orphan data deletion, f) storage load testing, etc., [17, 18].

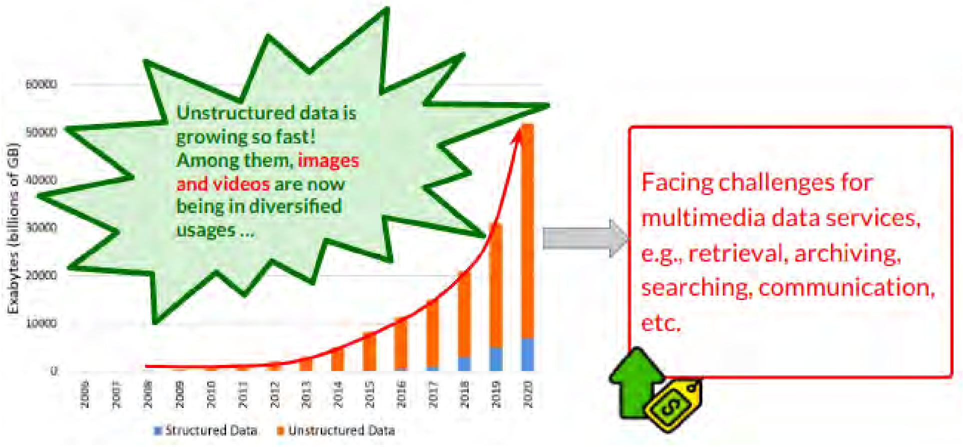


Figure 1.2: Structured and unstructured data growth over time [1]

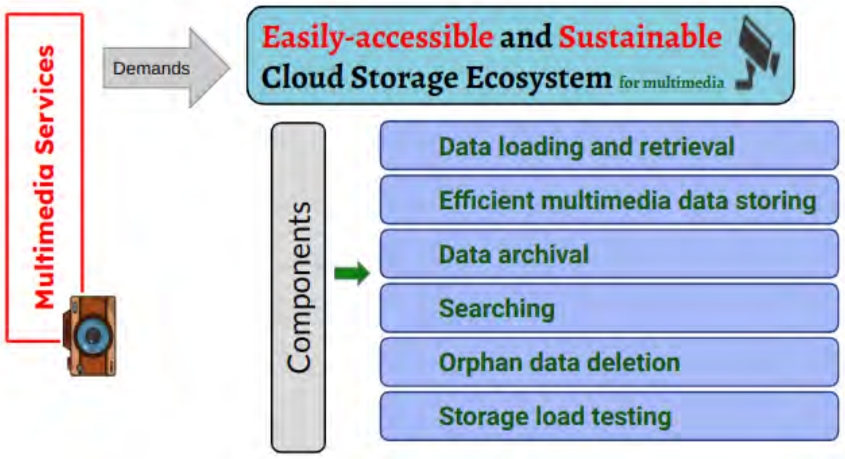

Multimedia Services
Demands
Easily-accessible and Sustainable Cloud Storage Ecosystem for multimedia
Components
Data loading and retrieval
Efficient multimedia data storing
Data archival
Searching
Orphan data deletion
Storage load testing

Figure 1.3: The demand for an easily-accessible and sustainable cloud storage ecosystem that can potentially grow without bounds or limits

1.1 Reserach Focus I: Image Loading and Retrieval from Cloud at Low-BW Context

Multimedia communication through cloud-based applications has become increasingly popular, due to the numerous benefits it provides. In order to access image data more efficiently and quickly, many of these systems utilize open-source projects. The JPEG format is the most commonly used method for storing images, and is supported by almost all image-capturing devices. In fact, as of 2015, over 7 billion images were produced daily in JPEG format. This number has increased significantly since then, with an expected 10.7% increase in the number of images stored in JPEG format from 2022 to 2023. JPEG uses the DCT (Discrete Cosine Transform) algorithm for lossy compression. The baseline method, which encodes all pixels sequentially, is the most widely used for JPEG operations as it offers the highest compression ratio and ensures the best image quality. Another method, called Progressive JPEG (PJPEG), is less commonly used. It loads lower frequency pixels of an image first, providing a lower quality preview of the image. Then, it refers to higher frequency pixels of the image, eventually displaying the complete image. This method is advantageous in bandwidth-constrained environments, as it allows for faster preview of the images.

Progressive image loading and retrieval has gained great interest in the research community in recent times which mostly explore Progressive images' performance from the perspective of high-bandwidth network connections [3, 19, 20]. The user experience has become very important in recent years, as the Internet traffic keeps increasing (more so due to the recent COVID-19 pandemic). Accordingly, to enhance the level of user experience, the performance of Progressive image loading and retrieval in a cloud environment needs to be improved even sustaining slow Internet connections and limited bandwidths [21] (in Figure 3.2).

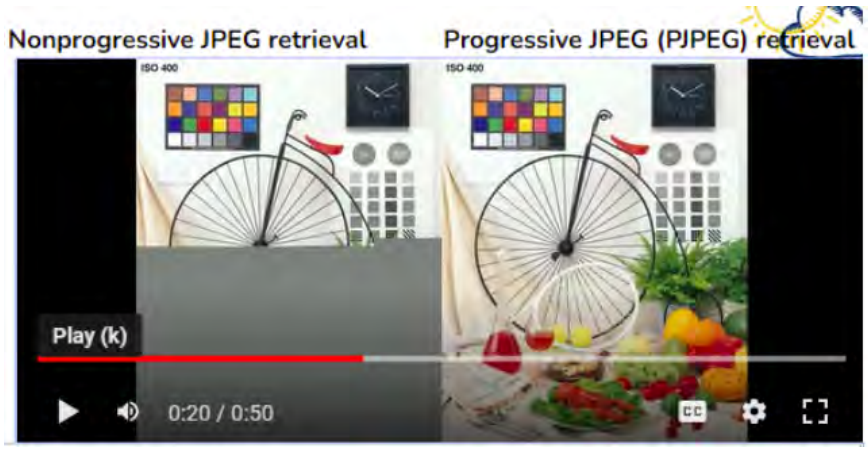

Nonprogressive JPEG retrieval
Progressive JPEG (PJPEG) retrieval
ISO 400
ISO 400
Play (k)
0:20 / 0:50

Figure 1.4: Image loading and retrieval from cloud at low-BW context

1.2 Research Focus II: Device-sensitive Multimedia Uploading, Retrieval, Searching, and Archival

The rise of embedded devices has created a significant challenge for media industries in terms of storing, processing, and managing vast amounts of data, including video, photos, audio, and text. This is due to the increasing production and consumption of such data by users through social media, online video, user-uploaded content, gaming, Software-as-a-Service applications, and other platforms. These applications all require accessible storage systems that can potentially scale infinitely without limitations.

With the advancement of technology, one important sector that is experiencing rapid growth in its field nowadays is the media retrieval and archival system specially for CCTV data [12,22,23]. Existing literature is yet to focus on an important realm of cloud-based video surveillance systems involving long-term storing and archiving [12]. In the CCTV data handling task, the video data needs to go through some processing to be ready to store in a cloud server with ease to ensure the availability and long term usage. Hence, we need to focus on storing large video files efficiently and archive the data for long-term purposes in a better manner enabling more space utilization.

Users upload different types of multimedia clips (to social media)
Need to access same multimedia files using diversified devices from remote locations
ring

Figure 1.5: Device-sensitive multimedia uploading, retrieval, searching, and archival

Every day, an enormous amount of data is created and stored on various cloud servers. Object storage is preferred due to its flexibility and consistency in storing such vast amounts of data. OpenStack is a distributed and consistent open-source cloud system that provides cloud object storage using an API called OpenStack Swift, which is scalable, durable, and designed to store unstructured data. Unlike file-based storage or block storage, Swift stores each piece of data as an object, making it an ideal storage system for handling massive amounts of data. However, retrieving and searching relevant data becomes challenging as the amount of data increases. The storage of consumer and business data in public or private clouds further complicates the efficient retrieval of meaningful data.

Recently, the problem of storing massive amounts of data is solved due to the complex architecture of object-based storage systems (S3, Swift, Swarm), still retrieving or searching for a certain object/file has become a major challenge [24–26]. Moreover, finding the real path using the linear searching method inside this storage is very time-consuming as the different replica copies are located in different regions [27]. Hence, Content-Based Search (CoBS) is starting to grow as the usage of data is increasing and metadata-based systems are struggling to work on a large amount of data.

1.3 Research Focus III: Storage Sustainability through Middleware Placement and Orphan Garbage Data Deletion

The management and communication of data in Object Storage Systems heavily rely on the design and placement of middleware in proxy or storage servers. To ensure data availability, multiple copies of big data are stored, and regular syncing and checking are necessary to detect bit rot and file degradation and ensure long-term preservation storage. Various studies have been conducted to ensure data storage sustainability and prevent undesirable consequences [28, 29]. Crowd management, real-time location-aware services, and medical systems require access to multimedia data from diverse remote devices. For instance, crowd management of millions of pilgrims is challenging and requires appropriate processing and communication from the cloud [30, 31]. Hence, emerging context-aware and location-aware cloud-based frameworks and services need both online and offline processing of unstructured data such as images and videos. Real-time video streaming is another significant feature of managing these services using cloud infrastructures.

Another important realm, efficient cloud side operation management [12] is needed for ensuring different features such as smooth video streaming, dynamic adaptive streaming, etc. Besides, proper and updated video segments need to be supplied from the cloud storage systems to achieve the features. Recent studies focus on several methods of mobile and web streaming [32, 33], gateway-based shaping methods for HTTP adaptive streaming (HAS), quality of experience of HAS, optimal transcoding and caching for adaptive streaming in content delivery networks [34], etc. In addition, offline processing (in Figure 1.5) of large multimedia data produces orphan data that have no information either in client side or in AUTH database, and the non-existence of information can occur due to network disconnection, client timeout problem, object versioning, etc., [35]. Research studies focus on several aspects of such redundant data deletion such as Linux container based deletion [36], Smartbin based deletion in wireless sensor networks [37], and assured deletion [38, 39] present some deletion approaches, which are not applicable to the case of orphan data in the cloud.

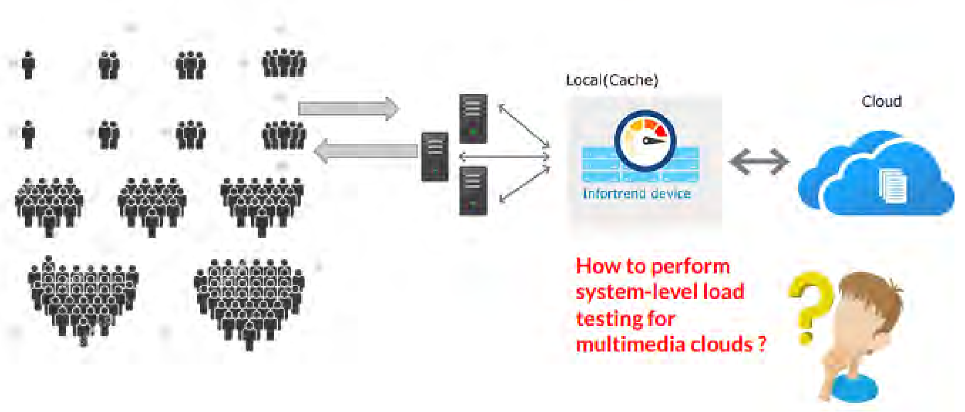

Local(Cache)
Infortrend device
Cloud
How to perform system-level load testing for multimedia clouds ?

Figure 1.6: System-level load testing of cloud storage ecosystem

1.4 Research Focus IV: System-level Load Testing of Cloud Storage Ecosystem

The demand for faster and easier access to data from connected systems is increasing, and designers need to test these distributed architectures under massive loads to ensure proper design. Service providers use caches to retrieve data faster from distributed private cloud systems, and analyzing the time elapsed for retrieving data from cache or backend is necessary to design a reliable system. Load testing is also needed to optimize software, hardware, and network parameters used in the system.

Currently, both private and public cloud service providers create their own distributed storage systems using multiple data centers. However, determining the optimal locations for deploying cache and backend cloud servers in these data centers is a challenging task. The time it takes to upload and download objects is directly affected by how the cache and backend servers are distributed.

Furthermore, for successful deployment of distributed architectures including caches and clouds in production environments, proper load testing is mandatory (in Figure 1.6). As such, several existing research studies focus on load testing tools and architectures based on performance and functional testing criteria. The study in [40] proposes empirical testing by monitoring user experience and system health in a feedback loop between traffic shifts. Other studies [41,42] propose automated approaches to validate whether a performance test resembles the field workload or not. Unfortunately, these studies propose and analyze only real-time test cases without focusing on network and software tuning using the outcomes of the test analysis.

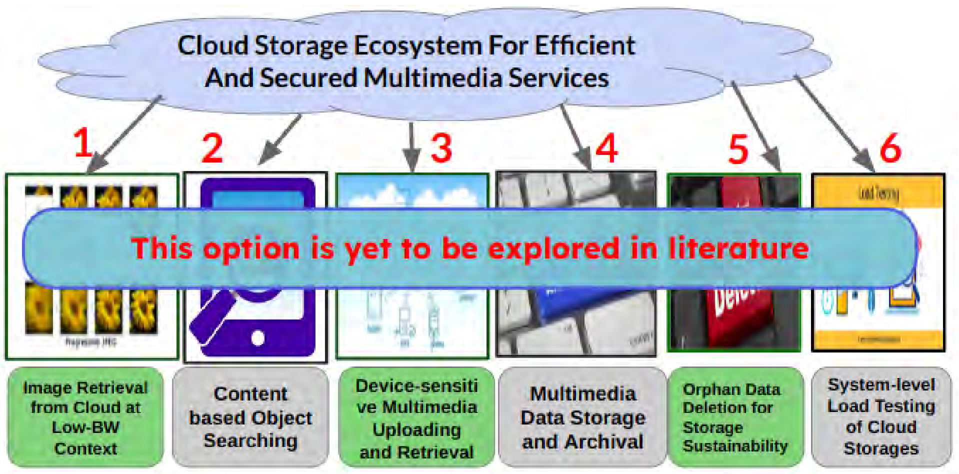

Cloud Storage Ecosystem For Efficient And Secured Multimedia Services
1
2
3
4
5
6
This option is yet to be explored in literature
Image Retrieval from Cloud at Low-BW Context
Content based Object Searching
Device-sensitive Multimedia Uploading and Retrieval
Multimedia Data Storage and Archival
Orphan Data Deletion for Storage Sustainability
System-level Load Testing of Cloud Storages

Figure 1.7: Motivation behind our study

1.5 Our Key Research Questions and Motivation Behind the Study

Till now, to the best of our knowledge, no research study focuses on this aspect (storage ecosystem) towards achieving better user experiences through performing faster Progressive image and multimedia data loading, storing, retrieval, searching, orphan data deletion, and proper system load testing even under bandwidth constraints (in Figure 1.7). Based on the challenges and limitation that are presented in the previous Sections, we design our key research question. Our key research question is -

"How can we design a novel cloud storage ecosystem for efficient and secured multimedia services?"

Besides, we dig down deeper based on the components of a storage ecosystem and formulate depended research questions for every components. They are -

- *How can we devise a methodology for efficient image retrieval from and storage to cloud even in low-BW context?*
- *How can we develop a new and faster content-based searching architecture for object storage systems?*
- *How can we design a private cloud system for device-sensitive multimedia uploading, retrieval, and long-time archival?*
- *How can we develop methods for finding orphan data in multimedia storage and to remove the orphan data in an efficient way to lessen system-level delay and overhead?*
- *How can we design case-driven system-level load testing methodology for cloud storage (considering both cache and backend)?*

Our work is motivated by the limitations of state-of-the-art research studies and the challenges involved in developing a novel cloud storage ecosystem. Here, we plan to focus on efficient and secured multimedia data retrieval and communication over the cloud storage. Accordingly, our goal is to design different system-level frameworks of the storage ecosystem, develop a private multimedia cloud, and devise load-testing methods. Thus, the specific objectives of this research study are as follows:

- To propose a novel cloud storage ecosystem for efficient multimedia data operations such as retrieval, searching, storing, archiving, communication, etc.
- To investigate retrieval of Progressive JPEG (PJPEG) architecture to optimize the scan scripts of PJPEG, and based on the findings of the investigation, develop a new lossy PJPEG architecture for faster image retrieval.
- To develop a new content-based searching architecture for object storage systems containing massive amounts of content.
- To devise a new mechanism for faster multimedia data storing and retrieval from the cloud storage, pertinent to several dimensions of remote devices.
- To devise an efficient multimedia data archival framework to store and archive large videos (such as surveillance data).
- To develop methods for finding orphan data in multimedia storage and to remove the orphan

data in an efficient way.

- To develop a new load-testing framework using cache and backend storage through network and software tuning.

1.6 Our Contributions in this Study

In this study, we propose a novel cloud storage ecosystem for efficient and secured multimedia services. Based on our work, we make the following set of contributions for proposing the ecosystem in this study:

- We investigate PJPEG loading, having DC and AC components, and develop a new Progressive Scan Script through optimizing the libjpeg library [43]. At first, we encode four DC coefficient data bits in the First Scan of libjpeg without degradation in the image quality. Furthermore, we propose a new lossy PJPEG architecture to reduce the image size and improve the scanning of PJPEG's picture payloads. The scanning is improved by using the change in bit pixels of distinct Luma and Chroma components.

- Next, we develop a content-based object searching architecture (using BERT, Darknet model, YOLOv4/YOLOv8 algorithm, and Elasticsearch) to extract additional metadata from images and keywords from documents. The extracted metadata is stored in a database that helps in searching for the desired data. In addition, a secured OpenStack Swift JOSS client user interface is created in order to access Swift and Elasticsearch clusters at the same time using user-level authentication tokens.

- Afterwards, a new mechanism for private cloud unifying object storage (Storage-as-a-Service) with cloud security, media processing (Processing-as-a-Service), and archiving media is proposed for covering diversified devices. we develop three new middleware services named 'PhotoPool', 'MediaBucket', and 'SecureCloud' to perform media processing, transcoding, and encryption-decryption tasks to make the system more secure, highly-scalable, and faster-accessible to end users.

- Besides, we propose a new archival framework for large videos (such as surveillance data) that stores data in multiple locations - local storage (for short-time), cloud storage (intermediate time), and archival storage (for long-time) - for ensuring improved retrieval, availability, and fault tolerance.

- Then, a new middleware 'VideoSegmenter' is designed for supporting HTTP adaptive streaming in object storage systems. We analyze the deployment of the middleware (either in a proxy server or in an object server) and improve it for avoiding extra overhead due to orphan data. Besides, a 'deletion daemon' named 'RemOrphan' is developed for removing the orphan garbage data using OpenStack rings and custom scripts.

- Moreover, we develop a new load-testing framework based on diversified real scenarios covering different types of protocols, URLs, loads, and servers using diversified tools such as JMeter, Ansible, and custom bash scripts.

- Finally, a performance evaluation of the components of the proposed ecosystem is done based on QoE (Quality of Experience) and QoS (Quality of Service). Performance metrics under consideration are: a) Quality of multimedia data, b) Size of resized images and videos, and c) Storing, downloading, and archival time. To evaluate the metrics, we perform the following tasks:

  - We deploy two different testbed setups (a local testbed in Bangladesh and a remote testbed in Toronto, Canada). In each of these setups, eight different servers are installed - two for Proxys, three each for Account and Container, and three for Objects. The reason behind having two different setups is to analyze how agnostic the proposed ecosystem is with respect to network latency.

  - For rigorous load-testing, one varnish cache server in front of the proposed Swift backend server and ten clients from different geographical locations are set up to evaluate the real scenarios.

  - System-level performance of different proposed frameworks is measured and compared with different alternative methods and existing studies. User-level evaluation is done by presenting converted images and videos of the proposed systems to different observers, and visual evaluations is analyzed using a standardized method [11].

1.7 Outline of this Study

This is how the rest of this study is organized.

- In Chapter 2, we present an overview of the architecture of OpenStack Swift and the QoE metrics based on which we perform our study. All the necessary components are briefly described in this Chapter.
- Then, we present the image loading and retrieval from cloud at low-BW context as our research focus I. Hence, in Chapter 3, we demonstrate the orchestrating of image retrieval and storage over a cloud system.
- Next, we present device-sensitive multimedia uploading, retrieval, searching, and archival (research focus II). At first, in Chapter 4, we illustrate secure processing-aware media storage and archival (SPMSA). Then we present a novel approach of content-based searching in object storage system in Chapter 5.
- After, we present storage sustainability through middleware placement and orphan garbage data deletion (research focus III). In Chapter 6, we demonstrate object storage sustainability through removing offline-processed orphan garbage data.
- Furthermore, we illustrate system-level load testing of cloud storage ecosystem (research focus IV). In Chapter 7, we present 'svLoad', an automated test-driven architecture for load testing in cloud systems.
- Finally, We present our future plan and conclude this study in Chapter 8.

Chapter 2

Background

Cloud storage systems play a crucial role in various aspects, as highlighted in the literature [44]. Firstly, they simplify the overall system by providing a centralized storage solution that can be accessed from anywhere. Additionally, cloud storage systems ensure data redundancy and backup by storing data across multiple servers, offering protection against physical threats such as hardware failures or disasters. This data redundancy also enhances data availability, allowing users to access their data at any time and from any location.

Moreover, cloud storage systems offer scalability and flexibility, providing room for growth as storage needs increase. They enable collaboration by allowing multiple users to access and share data simultaneously, facilitating seamless teamwork and data exchange. Figure 2.1 illustrates an environment for cloud computing [2], showcasing the various components and interactions within a cloud storage system. For the purpose of our study, we specifically focus on private cloud storage due to its practicality and applicability in different scenarios.

In this chapter, we present important background information relevant to our study [12]. We delve into storage evaluation techniques and provide an overview of OpenStack Swift, which is an open-source Object Storage System (OSS) widely used in cloud storage environments. Additionally, we explore various methods and approaches related to Quality of Experience (QoE) in the context of cloud storage systems. This comprehensive background information sets the foundation for our study and helps to establish a solid understanding of the key concepts and factors involved in cloud storage and its evaluation.

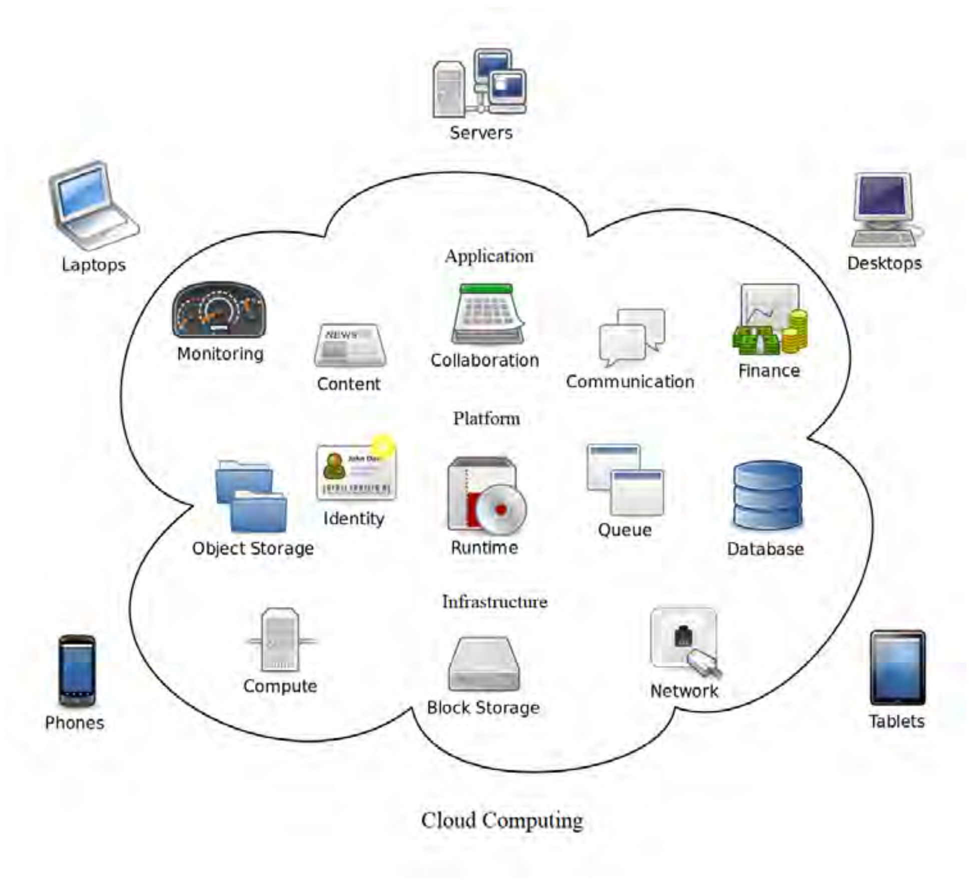


Figure 2.1: An architecture of cloud computing [2]

2.1 Overview of OpenStack Swift

In this section, we present a brief overview of a baseline architecture for secure and parallel processing of large-scale data stored in a cloud storage environment. The foundation of this architecture is built upon the principles of open-source software, which provides the freedom to modify and redistribute the source code. One such open-source platform that embodies these principles is OpenStack [3].

OpenStack is a widely used open-source platform that offers a comprehensive set of tools and services for building and managing cloud storage environments. It provides a flexible and scalable infrastructure for storing and processing massive amounts of data. OpenStack's modular architecture allows for customization and integration with other technologies, making it a versatile choice for cloud storage deployments.

The baseline architecture outlined in this section focuses on ensuring the security and efficient processing of data in a parallelized manner. By leveraging the capabilities of OpenStack, this architecture addresses the challenges associated with handling and analyzing large volumes of data stored in the cloud. It provides mechanisms for secure data storage, access control, and parallel processing techniques, enabling efficient and reliable processing of mass data.

Overall, this baseline architecture serves as a foundation for designing and implementing secure and parallel processing systems in a cloud storage environment. It takes advantage of the open-source nature of OpenStack to provide a flexible and customizable solution for organizations seeking to leverage the benefits of cloud storage while ensuring data security and efficient data processing.

OpenStack Swift is a widely adopted open-source cloud storage solution that provides a highly available, distributed, and consistent object/blob store for the OpenStack platform [3]. It functions as an object storage system, prioritizing eventual consistency over immediate consistency, which allows it to achieve high availability, redundancy, throughput, and capacity. One of the key features of Swift is its ability to store and manage a vast number of objects across multiple nodes. It employs a RESTful HTTP API, allowing users to interact with the storage system by submitting GET requests to download files and PUT requests to upload files [3].

In OpenStack Swift, a Proxy Server plays a crucial role in managing metadata. When a GET or PUT request is received, the server consults a ring structure to determine the locations of the account, container, or data object involved in the request. The ring serves as a mapping between the names of entities stored on disk and their physical locations [4]. Based on this information, the request is routed to the appropriate storage nodes. The architecture of OpenStack Swift is illustrated in Figure 2.2. It showcases the various components involved in the storage system, including the Proxy Server, the Ring, and the distributed storage nodes. This architecture enables Swift to provide a scalable and resilient storage infrastructure for storing and retrieving objects in a cloud environment.

Overall, OpenStack Swift offers a powerful and scalable solution for cloud-based object storage. Its focus on high availability and eventual consistency, combined with its distributed architecture, makes it well-suited for storing large amounts of data while ensuring data redundancy, accessibility, and performance. The architecture comprises several components as described in following sections:

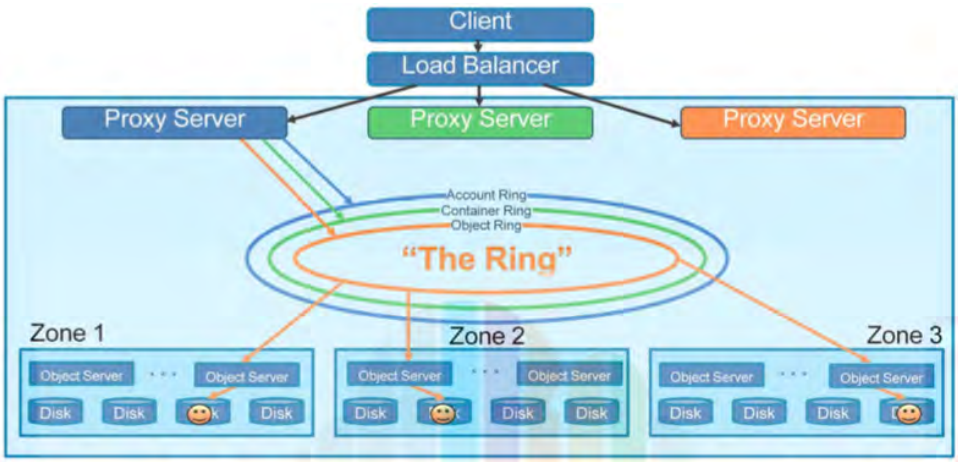


Figure 2.2: Overview of Swift architecture [3]

2.1.1 Proxy Servers

Proxy servers in OpenStack Swift play a crucial role in the storage system by handling incoming API requests and managing the routing of those requests to the appropriate storage nodes. Positioned after the load balancer, the proxy servers are responsible for determining which storage node should handle each request based on the URL of the objects. One important characteristic of the proxy servers is their shared-nothing architecture, which allows them to operate independently and scale horizontally as needed to accommodate varying workloads. This scalability is achieved by adding more proxy servers to the system when projected workloads increase.

Proxy servers also handle a significant number of failures. In the event that a server is unavailable for an object PUT operation, the proxy server will consult the ring structure to identify a handoff server, which can take over the responsibility of storing the object. This ensures that data is not lost or inaccessible in the face of server failures. Additionally, proxy servers play a role in coordinating responses and timestamps. They ensure that responses from different storage nodes are synchronized and consistent, and they assign timestamps to objects for tracking purposes.

Overall, proxy servers in OpenStack Swift are essential components that handle request routing, handle failures, enable scalability, and coordinate responses and timestamps, contributing to the efficient and reliable operation of the storage system.

2.1.2 Storage Servers

Storage servers are mainly account, container, and Object servers. Each type of storage servers are described in below subsections:

Object Server. The Object Server in OpenStack Swift is a straightforward yet important component responsible for storing, retrieving, and deleting objects within the storage system. An object represents the data to be stored, such as documents or images. Objects are stored as binary files on the local devices, utilizing the underlying file system of the object server. The object's metadata is stored in the file's extended attributes (xattrs), which provide additional information about the object. It should be noted that the chosen file system must support xattrs for proper functionality. Some file systems, like ext3, have xattrs turned off by default, requiring specific configuration.

Each object is stored using a path derived from the hash of the object's name and the timestamp of the operation. This path ensures unique identification and organization of objects within the file system. In case of conflicts, where multiple versions of an object exist, the "last write wins" policy is applied. This means that the most recent version of the object will be served. Deletion of objects is treated as a versioning process. A deleted object is represented by a 0-byte file with the extension '.ts', standing for "tombstone". This tombstone file ensures that deletions are correctly replicated across the system and that older versions of objects do not reappear due to failure scenarios.

By utilizing the Object Server, OpenStack Swift provides a reliable and efficient mechanism for storing, retrieving, and managing objects within the storage system, ensuring data integrity and consistency.

Container Server. A Container in OpenStack Swift represents a logical storage space where related objects are grouped together. The Container Server plays a crucial role in managing object listings within a container. Its main function is to handle the listings of objects, providing information about what objects exist in a specific container. Although the Container Server does not have knowledge of the precise locations of the objects, it maintains a record of the objects contained within the container. The listings, which contain details like object names, are stored as sqlite database files. Similar to objects, these listings are replicated across the cluster to ensure data redundancy and availability.

In addition to managing the listings, the Container Server keeps track of statistics related to the container. This includes monitoring the total number of objects stored in the container and tracking the overall storage usage. These statistics provide valuable insights into the container's contents and assist in effectively managing storage resources. By handling object listings and tracking essential container information, the Container Server plays a vital role in organizing and facilitating the retrieval of data within the OpenStack Swift storage system.

Account Server. An Account in OpenStack Swift defines the access rights and permissions for Containers and Objects. The Account Server, similar to the Container Server, fulfills the role of managing listings. However, its focus is on the listings of containers rather than individual objects. The Account Server maintains a database known as the Account database. This database contains a comprehensive list of all the Containers that are associated with a particular Account. These containers are distributed across the Swift cluster and are accessible to the specified Account based on the assigned permissions.

The Account database serves as a central repository of information about the containers available to an Account. It provides a means to organize and manage the containers associated with the Account within the distributed storage system. By handling the listings of containers and maintaining the Account database, the Account Server plays a critical role in managing access and facilitating control over containers within the OpenStack Swift storage environment.

2.1.3 Data Model in Swift

OpenStack Swift allows users to store unstructured data objects with a canonical name containing three parts: account, container, and object. Using one or more of these parts allows the system to form a unique storage location for data.

- **Account:** The account storage location is a uniquely named storage area that will contain the

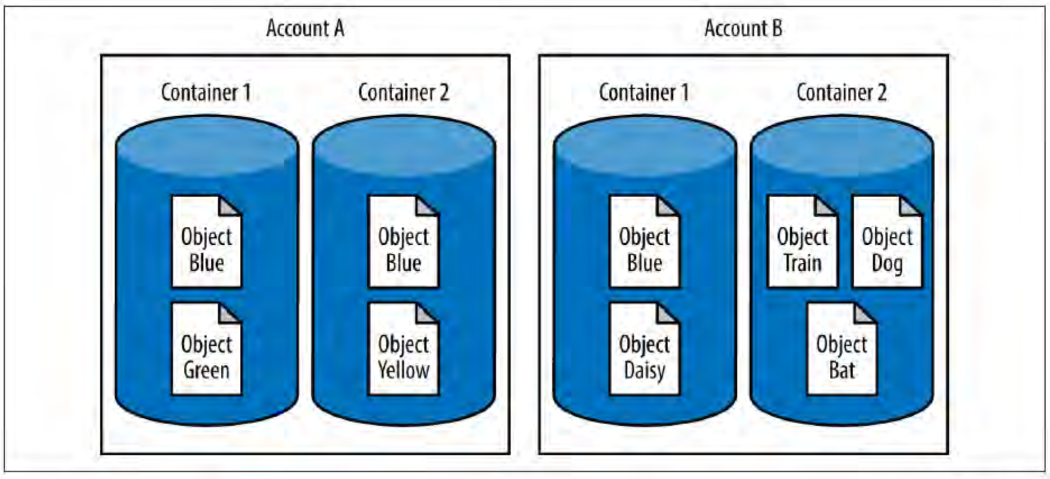


Figure 2.3: Objects can have the same name as long as they are in different accounts or containers [4]

metadata (descriptive information) about the account itself, as well as the list of containers in the account. In Swift, an account is not a user identity rather than storage area.

- **Container:** The container storage location is the user-defined storage area within an account where metadata about the container itself and the list of objects in the container will be stored.
- **Object:** The object storage location is where the data object and its metadata will be stored.

  If three objects with the name ObjectBlue are uploaded to different containers or accounts, each one has a unique storage location, as shown in Figure 2.3. The storage locations for the three objects are:
  /AccountA/Container1/ObjectBlue
  /AccountA/Container2/ObjectBlue
  /AccountB/Container1/ObjectBlue

2.1.4 Swift Architecture

Swift data, including accounts, containers, and objects, is ultimately stored on physical hardware. A node refers to a machine that executes Swift processes. A Swift cluster consists of multiple nodes that work together to perform all the necessary processes and services required for functioning as a distributed storage system. In order to enhance reliability and minimize the impact of failures, developers arrange the nodes within a cluster into regions and zones (Figure 2.4).

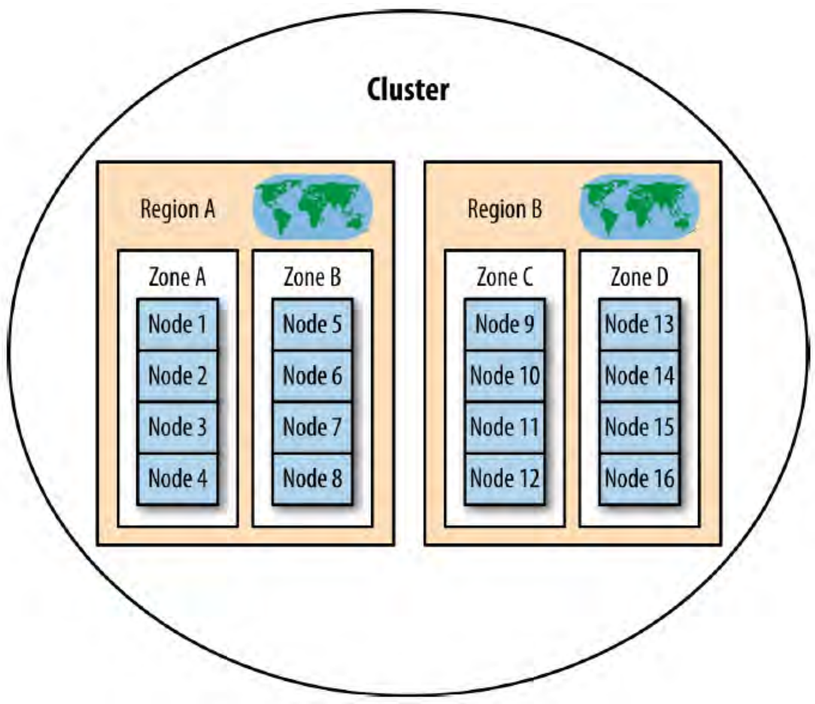


Figure 2.4: How nodes, zones, and regions are organized into a cluster [4]

The terms related to Swift architecture are:

- **Cluster:** A Swift cluster is the distributed storage system used for object storage. It is a collection of machines that are running Swift's server processes and consistency services.
- **Region:** Swift allows a physically distinct part of the cluster to be defined as a region. Regions are often defined by geographical boundaries; for example, several racks of servers (nodes) can be placed in higher-latency, off-site locations. When a multi-region cluster receives a read request, Swift will favor copies of the data that are closer, as measured by latency.
- **Zone:** A Zone represents a location that can isolate data. This could be a drive, a server, a cabinet, a switch, or even a data center.
- **Nodes:** Nodes are physical servers responsible for executing one or more Swift server processes. The primary Swift server processes include the proxy, account, container, and object processes. When a node runs an account or container server process, it also stores the corresponding account or container data and metadata. Similarly, a node running an object server process is responsible for storing objects and their associated metadata.

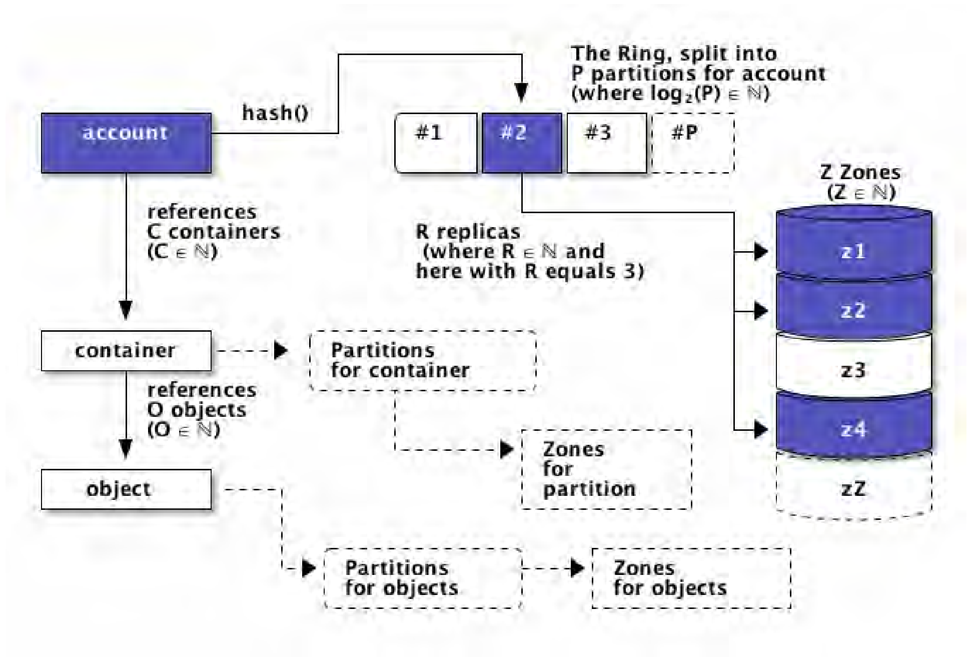

The Ring, split into P partitions for account (where $\log_2(P) \in \mathbb{N}$)
account
hash()
#1
#2
#3
#P
Z Zones ($Z \in \mathbb{N}$)
references C containers ($C \in \mathbb{N}$)
R replicas (where $R \in \mathbb{N}$ and here with R equals 3)
z1
z2
z3
z4
zZ
container
Partitions for container
references O objects ($O \in \mathbb{N}$)
Zones for partition
object
Partitions for objects
Zones for objects

Figure 2.5: Eventual consistency of OpenStack Swift using consistent hash ring [5]

2.1.5 Rings

The Ring is responsible for maintaining the mapping between logical data locations and their corresponding physical locations on specific disks. It is a static data structure that exists outside of the cluster. Devices within the Ring are assigned to partitions based on various policies such as regions, zones, and other constraints, ensuring fault tolerance and load balancing. Each Ring represents a mapping between the names of entities stored on disk (such as accounts, containers, and objects) and their physical locations. To perform any operation on an object, container, or account, other components in the system need to interact with the relevant Ring to determine its location within the cluster.

The Ring utilizes a customizable number of bits from the MD5 hash of an item's path to generate a partition index. This index determines the device(s) where the item should be stored. The number of bits retained from the hash is referred to as the partition power, and it defines the total partition count, which is equal to 2 raised to the power of the partition power. Partitioning the MD5 hash ring enables the cluster components to process resources in batches.

The Ring maintains the mapping of zones, devices, partitions, and replicas. By default, each partition in the ring is replicated three times across the cluster, and the ring's mapping stores the locations

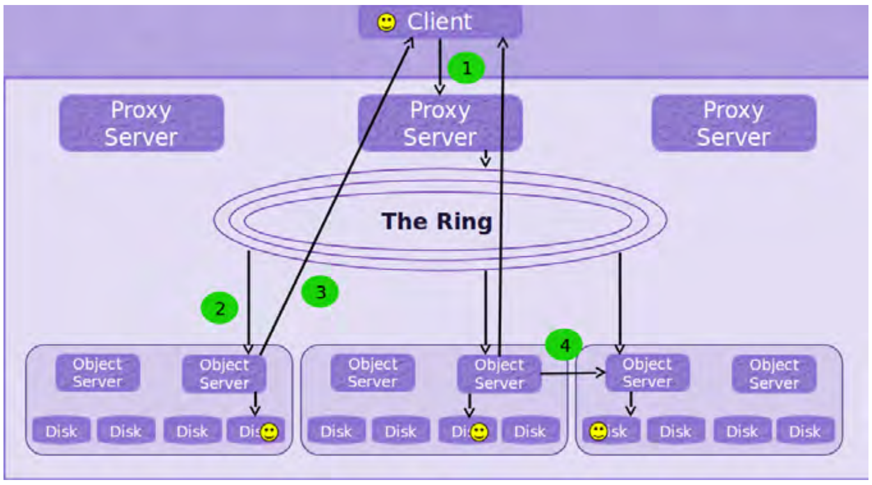


Figure 2.6: Execution of a PUT request in Swift [4]

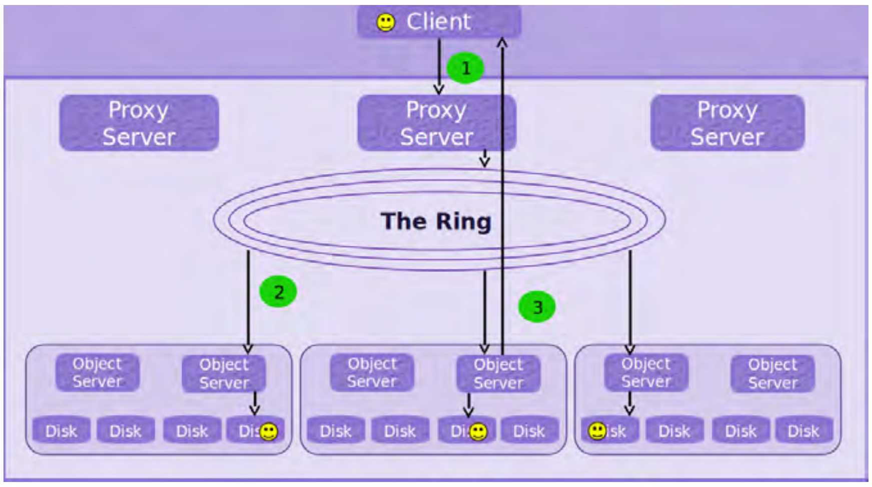


Figure 2.7: Execution of a GET request in Swift [4]

for each partition. In the event of failures, the ring also determines which devices should be used for handoff, ensuring data redundancy and fault tolerance. Other terms related to rings are:

- **Consistent hashing:** Swift uses the principle of consistent hashing for making ring. A ring represents the space of all possible computed hash values divided in equivalent parts. Each part of this space is called a partition. Fig. 2.5 presents the architecture of eventual consistency of OpenStack Swift using consistent hash ring [5].

- **Partition:** A Partition stores Objects, Account databases, and Container databases.

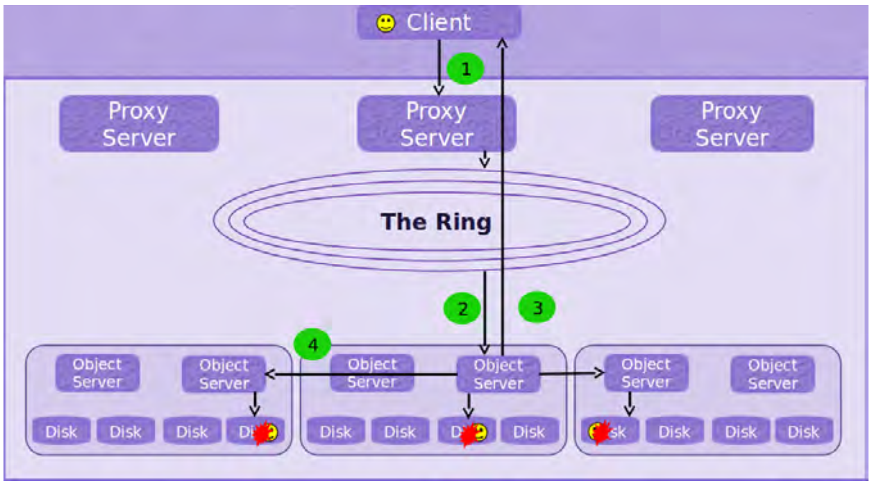


Figure 2.8: Execution of a DELETE request in Swift

2.1.6 Swift Consistency Process

In addition to the components, Swift has several consistency processes. the processes are as follows:

- **Auditor:** The Auditor process scans the disks at the same node to ensure that the stored data has not suffered any file system corruption.
- **Replicator:** The Replicator process ensures that enough copies of the most recent version of the data are stored where they should be in the cluster. The process also handles object and container deletions.
- **Account Reaper:** The Account reaper process locates an account marked as deleted and performs stripping out all objects and containers associated with the account. These actions ultimately removes the account records. To guard against error, the reaper can be configured with a delay so that it will wait for a specified period of time before it starts removing records.
- **Container and Object Updaters:** Container updater consistency process is responsible for keeping the container listings up-to-date in the accounts. Additionally, it updates the object count, container count, and bytes used in the account metadata. Besides, Object updater updates the container listing as well as the object count and bytes used in the container metadata.
- **Object Expirer:** This process allows designated objects to be automatically deleted at a certain time.

These components and processes of Swift can be used for performing different operations such as

PUT, GET, DELETE, etc. Fig. 2.6, 2.7, and 2.8 depict sequential steps for such operation. Details of these operations are given in [3].

2.2 Quality of Experience (QoE)

In order to achieve optimal quality of experience (QoE) for lossy image compression, developers must employ popular QoE techniques. Evaluating a user's QoE involves considering both objective and subjective factors. Objective factors encompass parameters at the network layer (such as jitter, packet loss, and delay) and the application layer (including resolution and frame rate) [45]. These objective factors determine the visual disparity between an image and its original definition, with resolution playing a significant role. However, subjective factors are more intricate in nature. Subjective factors extend to users' psychological conditions, preferences, and profile information (such as age and gender). For subjective and objective measurements of QoE, commonly used image quality metrics are [46, 47] as follows:

2.2.1 MOS

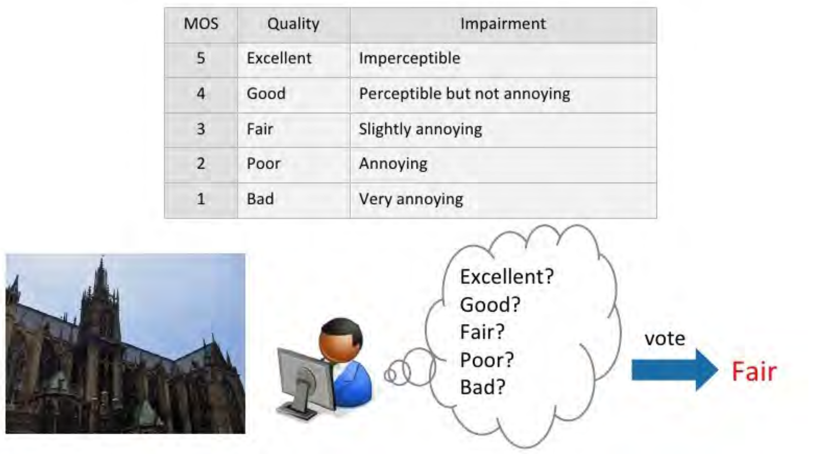
In accordance with ITU-T Recommendation P.910, the Mean Opinion Score (MOS) is selected as the scoring criterion for subjective QoE measurement, representing the assessment provided by a test panel. In the realm of multimedia communication, the perceived quality typically determines whether the experience is considered good or bad. Alongside qualitative descriptions such as 'good' or 'very bad', there exists a numerical method for expressing quality, known as the Mean Opinion Score (MOS). MOS provides a numerical indication of the perceived quality of transmitted and compressed media after being processed by codecs.

MOS is expressed in one number, from 1 to 5, 1 being the worst and 5 the best. MOS is quite subjective, as it is based on figures that result from what is perceived by people during tests. However, there are software applications that measure MOS on networks. The MOS is expressed on a five-point scale (in Figure 2.9), where-

- 5 = excellent
- 4 = good
- 3 = fair

- 2 = poor and
- 1 = bad

. The minimum threshold for acceptable quality corresponds to a MOS of 3.5 [48]. Due to the human tendency to avoid perfect ratings (now reflected in the objective approximations), somewhere around 4.3 - 4.5 is considered an excellent quality target.



MOS	Quality	Impairment
5	Excellent	Imperceptible
4	Good	Perceptible but not annoying
3	Fair	Slightly annoying
2	Poor	Annoying
1	Bad	Very annoying

Figure 2.9: MOS calculation depends on an user [6]

2.2.2 PSNR

Peak-Signal-to-Noise-Ratio gives the ratio (in dB) between power of the original signal and power of a reconstructed compressed signal. PSNR is usually derived via mean squared error (MSE) between two signals in relation to the maximum possible luminance of images. MSE and PSNR are calculated as [46, 47]:

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - K(i,j)]^2 \tag{2.1}$$

$$PSNR = 10 \log_{10} \left(\frac{{MAX_I}^2}{MSE} \right) \tag{2.2}$$

Equation 4.1 and 2.2 present equations of MSE and PSNR of a noise-free $m \times n$ monochrome image I and its noisy approximation K. Here, MAX_I is the maximum possible pixel value of the image. Although PSNR may not always accurately reflect the QoE, as demonstrated in [49], it continues to be a popular method to evaluate quality difference among videos.

2.2.3 SSIM

Structural Similarity Index (SSIM) is a perceptual metric that quantifies image quality degradation caused by processing such as data compression or by losses in data transmission. It is a full reference metric that requires two images from the same image capture — a reference image and a processed image (in Figure 2.10).

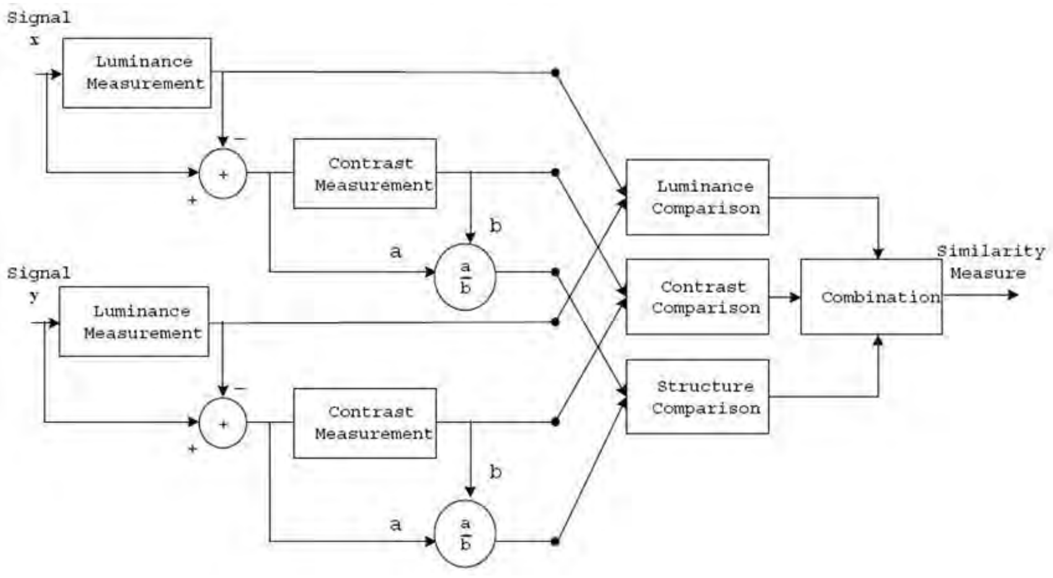


Figure 2.10: Structural Similarity Index measurement [7]

Structural Similarity Index [50] uses a structural distortion based measurement approach. Structure and similarity in this context refer to samples of the signals having strong dependencies between each other, specially when they are close in space [51]. Here, the rationale is that human vision is specialized in extracting structural information from the viewing field, not in extracting errors.

2.2.4 VQM

The Video Quality Metric, as referenced in [52], is a measurement that assesses the perceptual impact of various image impairments in videos. These impairments include blurring, jerky or unnatural motion, global noise, block distortion, color distortion, and combinations thereof. Video quality is a characteristic of a video signal as it traverses a transmission or processing system, quantifying the perceived degradation in comparison to the original source video. When video processing systems are employed, they may introduce certain distortions or artifacts into the video signal. However, the extent of these distortions can vary depending on the complexity of the content and the selected parameters for processing.

The extent of degradation in video quality can vary, and whether it is perceivable or acceptable to an end user depends on individual preferences. Determining a universally acceptable quality level for all users is challenging, but it remains an important goal in video quality evaluation studies. It is crucial to understand the various types of visual degradations or artifacts in terms of their annoyance factors and to evaluate the perceived quality of a video from the end user's perspective.

The Video Quality Metric (VQM) is also utilized for comparing two interpolated images. In summary, higher Mean Opinion Score (MOS), higher Peak Signal-to-Noise Ratio (PSNR), higher Structural Similarity Index (SSIM), and lower Video Quality Metric (VQM) values indicate better quality for multimedia data, such as images and videos.

Part I: Image Loading and Retrieval from Cloud at Low-BW Context

Chapter 3

Orchestrating Image Retrieval and Storage over A Cloud System

3.1 Introduction

With so many applications, multimedia communication over the cloud is gaining significant interest in recent times [11, 53]. These systems often leverage various open-source projects for faster and storage efficient access to image data, which is a critical component in multimedia communication over the Internet today.

There exists many formats to store images, among which JPEG is the most popular [54]. JPEG is used by almost all image-capturing devices today. In 2015, 7 billion images were produced in JPEG format every day [55], which is much higher now. The number of images stored in JPEG format from 2022 to 2023 is expected to increase by 10.7%. Besides, 74.2% of the websites use JPEG as their image format. Thus, as there is a huge amount of data stored in this format, and as such, optimizing retrieval of JPEG images is of utmost significance today.

JPEG performs lossy compression using an algorithm called DCT (Discrete Cosine Transform). For performing JPEG operations, baseline method is mostly used. This method works by encoding all the pixels sequentially. It produces the highest compression ratio and guarantees the best image quality. A less used method is Progressive JPEG (PJPEG). It works by loading lower frequency pixels of an image (or a low-quality presentation of the image) first. Later, it refers to the higher frequency pixels

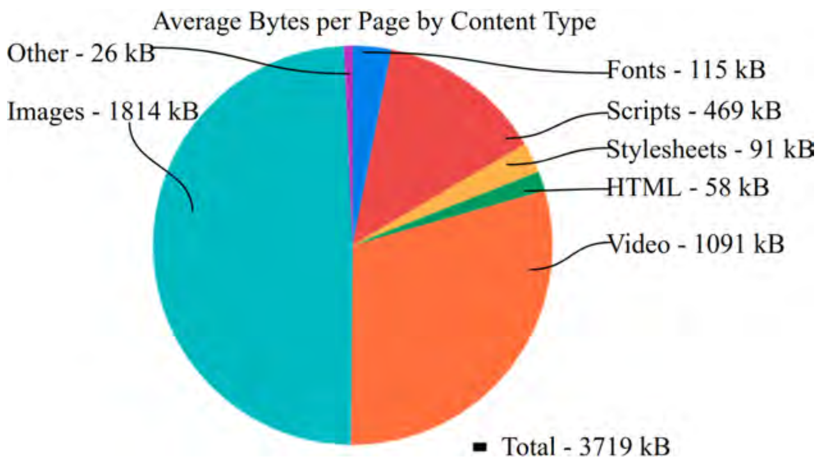


Figure 3.1: Average bytes per page by content type [8]

of the image. It shows a faster preview of the images. Hence, Progressive JPEG offers advantages under bandwidth-constrained environments.

Investigating the notion of Progressive image loading and retrieval has gained great interest in the research community in recent times [19]. Research studies [21, 56] in this regard mostly explore Progressive images' performance from the perspective of high-bandwidth network connections. However, slow Internet connections and limited bandwidths are a reality in many countries all over the world [57]. The user experience has become very important in recent years, as the Internet traffic keeps increasing (more so due to the recent COVID-19 pandemic [58]). Accordingly, to enhance the level of user experience, the performance of Progressive image loading and retrieval in a cloud environment needs to be improved even sustaining slow Internet connections and limited bandwidths.

In the case of Progressive Image Retrieval, the existing method for encoding PJPEG consists of loading 7 DC coefficient bits in the First Scan [14]. As the DC coefficient (pixel) usually contains high-magnitude values, it takes substantial time to load the 7 bits. To decrease the loading time, one way is to load a lower number of DC coefficient bits that can provide a solution for Faster Image Retrieval. Till now, to the best of our knowledge, no research study focuses on this aspect towards achieving better user experiences through performing faster Progressive image loading even under bandwidth constraints. Besides, existing research studies are also yet to focus on this important realm in multimedia cloud operation and communication covering Faster Image Retrieval using progressing schemes sustaining the bandwidth limitation. This is equally applicable to popular OpenStack-like systems such as SPMS (Secure Processing aware Media Storage) [11].

To address this, in this Chapter, we propose a new Progressive Scan Script using fewer bits in the First Scan. We encode only 4 DC coefficient data bits in the First Scan without degradation in the image quality. Hence, it shows a much faster visualization of the image. User Waiting Time significantly decreases to 54% after using our new Script. A potential downside of the Scan Script is that it tends to make image size larger. Hence, we also propose a PJPEG lossy Architecture to overcome the drawback by reducing image file size.

Based on our study, we make the following set of contributions in this Chapter:

- We propose a new Scan Script for Faster Image Retrieval. Our proposal is inspired by a thorough investigation of the open-source libjpeg library [43] and optimization of scan scripts for Progressive JPEG.
- To overcome a potential downside of our proposed Scan Script of making image size larger, we propose a new lossy PJPEG architecture to produce smaller-sized image files.
- We implement our proposed architecture in a real testbed comprising a high-configuration server in Canada and a client in Bangladesh, which embraces the notion of a private cloud. Besides, our testbed setup realizes limited bandwidth and slow Internet connection perspectives. In the process of implementing the testbed, we elaborate system design and deployment details of the proposed architecture.
- We conduct rigorous experimentation over the testbed setup to evaluate the performance of our proposed architecture. We compare our experimental results against that of alternative solutions over various devices. The comparison confirms the better performance of our proposed architecture compared to that of the existing alternative solutions.
- Further, we compare the performance of our proposed work with that of other state-of-the-art cloud applications such as Dropbox and Google Drive. Our results demonstrate superior performance than the default image loading methods of the state-of-the-art cloud applications. Nonetheless, we also compare advantages of our proposed approach compared to other recent state-of-the-art research studies.

3.2 Related Work

Fetching large images from public storage systems to own processing systems and then processing those images in the own processing systems — both appear to be expensive and time-consuming. Our previous work [59] focus to retrieve image efficiently and securely in a private cloud. We integrate resizing and encryption–decryption algorithms as a secured proxy service combined with a cloud file sharing environment named Swift. High-resolution images often take a substantial amount of time to load with average network bandwidth speed. In cases, it even considerably takes longer on mobile devices over wireless connections. Hence, many research studies focus on partial visual contents for better user experience. Study [20] presents CBIR (Content-Based Image Retrieval) system that achieves coarse-to-fine progressive RS (Remote Sensing) image description and retrieval in the partially decoded JPEG-2000 compressed domain. Study [60] proposes a cloud-based face video retrieval system with deep learning. Studies [61, 62] proposed a progressive image transmission scheme based on strategic decomposition and block truncation coding, respectively.

It is now popular to access images progressively [63]. A study of over 10,000 JPEG images from all over the web reveals that images of file size 10KB or higher have a better chance of being smaller when the Progressive JPEG method is used [64]. Studies [21, 56] present results from image-loading experiments that offer quantitative comparisons between common loading methods. Finally, they suggest a simple spiral variant. Open-source library libjpeg [43] contains the Scan Scripts. Scan Script is mainly responsible for progressive image loading. Besides, another study [65] focuses on field devices that rely on battery power to further economize on data transmissions. They do it to prolong deployment duration with particular use cases in wireless sensor networks. Moreover, study [66] approaches for tackling energy-beneficial VSN (Visual Sensor Networks) constraint problems include adapting JPEG by exploiting the DCT (Discrete Cosine Transform) energy compaction property. Their exploitation is performed by processing only a portion of each block of 8×8 DCT coefficients of the captured images using the global and local methods.

They propose that transmitting a subset of image data could potentially enhance the battery life of power-constrained devices. Such kind of progressive refinements exists in applications ranging from telemedicine, security, and surveillance where an initial assessment can lead to further exploration of only a small region. Hence, they propose to select minimum information for a coarser reconstruction by transmitting only the DC coefficients as the first or base layer. After, they will transmit more data

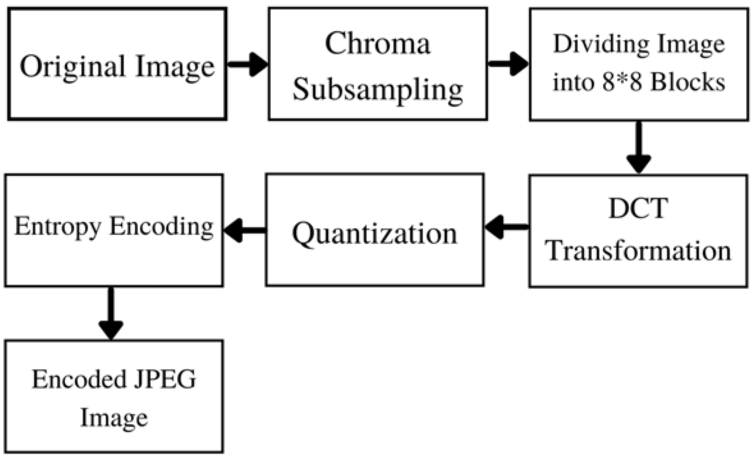


Figure 3.2: JPEG encoding technique

representing an entire image or a selected region-of-interest (RoI).

To the best of our knowledge, our proposed methodology is the first to focus on faster and smoother progressive image retrieval for a bulk amount of images even in the presence of bandwidth-constrained scenarios. As we have discussed previously, to overcome our scan scripts drawback, we work with PJPEG compression. Researchers have always been trying to make JPEG compression more efficient in many different ways [67]. Study [68] proposes reducing redundant data in the DCT domain by performing selective quantization and optical encoding for Baseline JPEG. Study [69] suggests image pre-processing steps to improve standard JPEG compression ratio by increasing color repetition probability. Study [70] modify JPEG based on quick DCT that removes the majority of zeros. Moreover, Study [71] propose to use segmented entropy encoding. Lastly, study [72] shows that dynamic resizing with progressive JPEG saves 2.5× read data over baseline JPEG at a Peak Signal-to-Noise Ratio (PSNR) of 32 dB.

3.3 Background

JPEG compression is a lossy compression. JPEG deletes data bits while performing different processes like chroma subsampling, quantization, entropy encoding, etc. In Fig. 3.2, we see the encoding process of JPEG compression. To start, JPEG turns images from RGB to a different color space named $YCbCr$. JPEG uses this color space to delete specific data bits. Y or luminance is the light intensity. Cb and Cr represents red chrominance, and blue chrominance respectively. Our eyes are more sensitive

Table 3.1: The order to scan DCT coefficients [13]

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

to luminance. Whereas, less sensitive to sudden changes in chrominance components [9, 13]. Fig. 3.3 shows changes in components after subsampling by 30%. Our eyes cannot detect sudden changes in chrominance. Hence, JPEG divides only the chrominance information by a factor of 2. This process is called chroma subsampling.

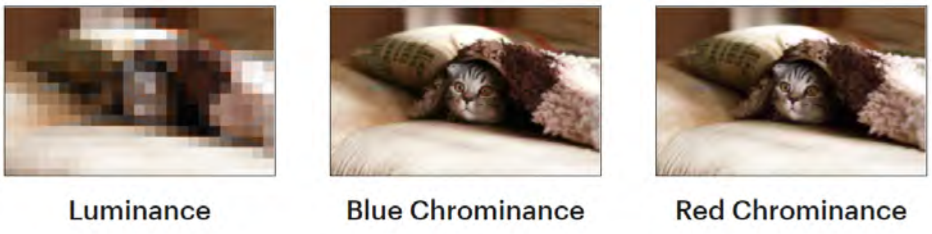


Figure 3.3: Subsampling by 30% [9]

Next, JPEG divides a picture into chunks of 8×8 blocks. Sequence for pixels in a 8×8 is shown in Table 3.1. Every block contains $64(0-63)$ pixels and every pixels consist of 3 components(Y, Cb, Cr). Pixel values are from 0-255. JPEG subtracts every pixel value by 128.

Later, JPEG uses DCT to convert 8×8 block components to a frequency domain.

$$\begin{aligned} & F(u,v) = \tfrac{1}{4}C(u)C(v)\textstyle\sum_{x=0}^{7}\sum_{y=0}^{7} \\ & f(x,y)\cos\left[\tfrac{\pi(2x+1)u}{16}\right]\cos\left[\tfrac{\pi(2y+1)v}{16}\right] \\ & \text{for } u = 0,\ldots,7 \text{ and } v = 0,\ldots,7 \\ & \text{where } C(k) = \begin{cases} \frac{1}{\sqrt{2}} \text{ for } k = 0 \\ 1 \text{ otherwise} \end{cases} \end{aligned} \tag{3.1}$$

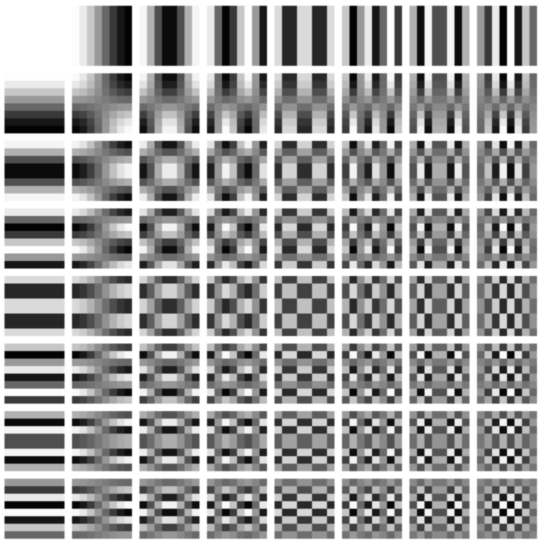


Figure 3.4: Discrete Cosine Transform (DCT) [10]

Table 3.2: The quality factor [13]

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Equation 3.1 [73], [74] represents DCT. JPEG gets 64 new coefficients or pixel values after using DCT for all of the components. The First Coefficient of a block represents the DC coefficient. This coefficient shows the general intensity of the whole image block. AC coefficients change the intensity and have a much less magnitude than the DC coefficient.

In Fig. 3.4, we see, from the DC coefficient, as we go horizontally by moving right or vertically by moving down to AC coefficients, the frequency keeps increasing. DC coefficient has much more effective than AC coefficients as our eyes are not good at differentiating high-frequency data bits.

JPEG further reduces these coefficients by dividing these coefficients by quantization matrix. Quantization matrix values are lower for DC and its closer AC coefficients. There are separate quantization matrix tables for luminance and chrominance. In Table 3.2, we see the quantization table for luminance. As shown In Equation 3.2, JPEG only preserves the rounded values after the division. The data we lost in the process of rounding value is not renewable. That is why JPEG is a lossy compression. This process is called quantization. Quantization helps to get lower values for high-frequency AC coefficients.

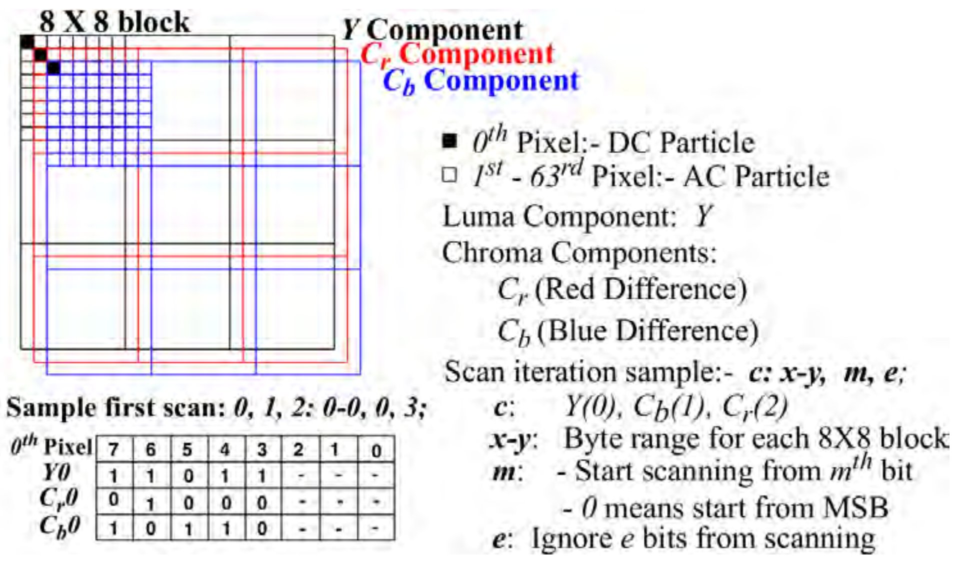
$$F_q(u, v) = \text{Round}\left(\frac{F(u, v)}{Q(u, v)}\right) \tag{3.2}$$

The last step for encoding JPEG is entropy encoding. Entropy encoding encodes coefficients with the same values in a zigzag format. The zigzag format is helpful to encode the image from a lower frequency to higher frequency data bits. Normally, Huffman Coding is used for entropy encoding. To decode the image, the processes are done again reversely.

The baseline method and the Progressive method encode pixels differently. The baseline method encodes images block by block. Where Progressive JPEG encodes specific pixels for every block script by script. Many social sites and websites are now using compressed and resized JPEG files to cover diversified remote devices [11, 53]. Hence, we briefly present the library of JPEG (libjpeg) [43] and OpenStack Swift-like media storage systems to provide a background related to our approach.

Libjpeg: Libjpeg library (written in C) is used in many platforms for handling JPEG image data format through implementing JPEG codec (encoding and decoding). It performs conversions between images inserting and exerting textual comments and transforming JPEG files using libjpeg-turbo [43].

SPMS (Secure Processing aware Media Storage): Recently, many media cloud storage such as SPMS are deployed using OpenStack Swift. Swift is an open-source object storage system having some special features. Such as eventual consistency, high availability, fault tolerance, replication, etc. It has two types of servers-proxy for management and processing and 3 storage servers (account, container, and object) for storing database and data objects [3]. Besides, the SPMS system has some special features of media securing, image data conversion to PJPEG, image resizing, video transcoding and resizing to various sizes, etc [11]. As SPMS-like media storage systems are used for multipurpose media management tasks (Such as video streaming and storing many versions of images), optimizing multimedia retrieval comes into play.


8 X 8 block
Y Component
C_r Component
C_b Component
■ 0th Pixel:- DC Particle
□ 1st - 63rd Pixel:- AC Particle
Luma Component: Y
Chroma Components:
C_r (Red Difference)
C_b (Blue Difference)
Scan iteration sample:- c: x-y, m, e;
c: Y(0), C_b(1), C_r(2)
x-y: Byte range for each 8X8 block
m: - Start scanning from mth bit
- 0 means start from MSB
e: Ignore e bits from scanning
Sample first scan: 0, 1, 2: 0-0, 0, 3;

0th Pixel	7	6	5	4	3	2	1	0
Y0	1	1	0	1	1	-	-	-
C_r0	0	1	0	0	0	-	-	-
C_b0	1	0	1	1	0	-	-	-

Figure 3.5: An example of 8 x 8 block JPEG structure of Luma and Chroma components. Here, the scan iteration sample is explained for progressive JPEG type images.

3.4 System Design and Implementation

We evaluate the performance of our proposed architectures through a real implementation. First, we briefly present our experimental testbed setup. Later, we present experimental results and findings for our architectures. Lastly, we compare our method with other existing studies.

3.4.1 Faster Image Retrieval

To ensure Faster Image Retrieval, partial loading is essential. Since Progressive JPEG allows partial encoding and decoding, we use Progressive JPEG. A Progressive JPEG is loaded Scan by Scan. The First Scan sets the parameter for the number of bits it will encode in the first partial loading. Hence, the less bit we use in the first Scan, the faster we load the first partial image. However, loading fewer bits can produce bad image quality. Our target is to encode a minimal number of bits for the first Scan while maintaining the visual quality same as the default Scans produced image.

For a better understanding of the architecture, here, we first present the structure of JPEG images in Fig. 3.5. The 0th pixel contains DC particle or coefficient and 1st to 63rd pixels contain AC coefficients [56]. Scan iterations over the pixels are represented with some variables. For example, each Scan Script can be represented by $c : x - y, m, e$. Here, c : $0, 1, 2$ (0 : Y component, 1 : C_r component, and 2 : C_b component). $x - y$ represents the pixel range that needs to be scanned for each 8 x 8 block. Thus, $0 - 0$ means scanning 0^{th}th pixel for each block. Additionally, m : refers to Scan 'm' last bits, i.e., bits after this index need to be scanned. Here, '$0'$ refers to the beginning or MSB. Nonetheless, e refers to skip 'e' bits counting from LSB. In Fig. 3.5, we present the sample of 0^{th}th pixel for Y component. Here, we select all the DC coefficients.

We have already seen in Fig. 3.5 how a scan works, Now, we break down the Scan Scripts. Table 3.4 for instance, contains a total of 17 Scans. The First Scan is $0, 1, 2 : 0 - 0, 0, 7$. It instructs the libjpeg library to encode only the first pixels $(0 - 0)$ of the luma and chroma components $(0, 1, 2)$. Encoding starts from the MSB and excludes 7 LSBs $(0, 7)$. Afterwards, the second Scan $(0, 1, 2 : 0 - 0, 7, 6)$ tells to start scanning from 7^{th}th MSB and ignore last 6 LSB's. Therefore, it completes scanning all the bits from the first bytes $(0 - 0)$ for each component in the 8^{th}th Scan. In 9^{th}th Scan $(0 : 1 - 27, 0, 1)$, it scans only the luma components from each blocks. It instructs to encode pixel 1 to pixel 27. Encoding starts from 0^{th}th MSB . It skips the last LSB. In the next two Scans: 10^{th} $(2 : 1 - 27, 0, 1)$ and 11^{th} $(1 : 1 - 27, 0, 1)$; it encodes the red chrominance and next blue chrominance correspondingly. Human eyes are more sensitive to red particles than to blue ones. Therefore, we firstly encode red chrominance and then the blue chrominance. Afterwards, 12^{th}th Scan $(0 : 28 - 63, 0, 1)$, 13^{th}th Scan $(2 : 28 - 63, 0, 1)$ and 14^{th}th Scan $(1 : 28 - 63, 0, 1)$; encode first 7 MSBs $(0, 1)$ for the remaining pixels $(28 - 63)$ for luma (0), red (2) and blue (1) components respectively. Final 3 scans: 15^{th} $(0 : 1 - 63, 1, 0)$, 16^{th} $(2 : 1 - 63, 1, 0)$ and 17^{th} $(1 : 1 - 63, 1, 0)$; encode the remaining last LSB $(1, 0)$ for all of the remaining pixels (1^{st} to 63^{rd}) for luma, red and blue components correspondingly. Finally, it completes all of the sequential rounds for the progressive conversion of the given baseline image. In the later Scan approaches, from Table 3.5 to Table 3.11, we have reduced the exclusion bit count in the first Scan. It results in loading more bits. This reduced exclusion of bit counts, making the images bigger in size even in the first Scan. From the corresponding tables, we see that the size for the corresponding images increases eventually after each round of the progressive Scans. At the end of the last Scan, it gets its full size.

Default Scan Script iterations are available online.[1]. First Scan of Default Scan Script is $0, 1, 2 : 0 - 0, 0, 1$. Hence, the default Scan Script encodes 7 bits from the DC coefficient for all three components in the First Scan. Our target is to encode the lowest number of bits for the First Scan with maximum visual quality. Hence, we make eight different Scan Scripts (SS_1 to SS_8) by increasing bit by bit

[1] *https://github.com/libjpeg-turbo/libjpeg-turbo/blob/1.0.x/jcparam.c* (Line No. 508-526)

gradually for the First Scan. For example, we encode 1 bit from DC coefficient in the First Scan of SS_1, 2 bits for SS_2, and 8 bits SS_8, etc. For maximum visual quality, we select both luma (or luminance) and chroma (or chrominance) components; otherwise, the First Scan will be only black and white.

The First Scan for Scan Script 1 (SS_1) is $(0, 1, 2 : 0 - 0, 0, 7)$, the First Scan for Scan Script 2 (SS_2) is $(0, 1, 2 : 0 - 0, 0, 6)$, the First Scan for Scan Script 8 (SS_8) is $(0, 1, 2 : 0 - 0, 0, 0)$, etc. Out of these 8 Scan Scripts, to challenge the default Scan Script, we need a script that encodes fewer bits in the first Scan and produces image quality the same as the default Scan Script's First Scan. Our proposed First Scan Script will be as follows:

$$SS_{s1} = \min_{i \in A} (V_{qi+1} - V_{qi}), \text{ where A =\{j\} such that } SS_{zj} \leq \frac{SS_{zj-min} + SS_{zj-max}}{2} \tag{3.3}$$

We find that encoding less than 4 bits in the first Scan does not produce the image quality we want. Again, encoding more than 4 bits does not make our image quality better. We represent SS as the Scan Scripts and SS_s as the Scan number of the Scan Scripts. SS_{s1} represents the first Scan of the Scan Scripts. SS_{zj} represents the size of the image using the first Scan. V_{qi} represents the visual quality of the i^{th} Scan.

However, there is a drawback to this architecture. The lower number of bits in the first scan results in a higher number of scan iterations. For example, for the SS1, we have 17 scan iterations, where our default scan script has only 10 iterations. It creates an increased file size for the corresponding image. To solve this issue, in the next subsection, we are proposing a new compression architecture for Progressive JPEG. We will present the performance evaluation in an SPMS-like system.

Scan Script 4(SS_4) encodes 4 bits in the first scan. We propose to use SS_4 rather than the default Scan Script. SS_4 have 14 Scans where the first Scan is $0, 1, 2 : 0 - 0, 0, 4$. Since we are trying to work on faster image retrieval, we focus on the first Scan; several Scans are not important.

A higher number of bits take more time to encode. we propose to encode only 4 bits rather than 7 bits of the DC coefficient in the first Scan for each component (Y, C_b, C_r). Hence, the user will be able to load the first Scan faster. Therefore, the first Scan (SS_{s1}) of Scan Script 4 (SS_4) is much faster than the default Scan Script.

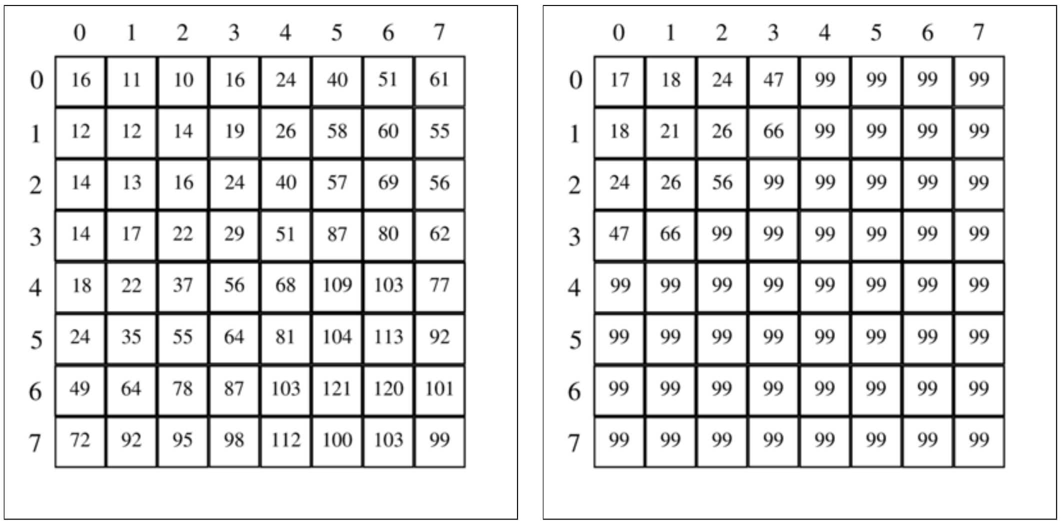
3.4.2 Lossy PJPEG Architecture

JPEG compression is a lossy compression. In JPEG, many databits from the original image are deleted. JPEG deletes data bits while performing different processes like chroma subsampling, quantization, entropy encoding, etc. In Fig. (3.2), we see the encoding process of JPEG compression. To start, JPEG turns images from RGB to a different color space named YC_bC_r. JPEG uses this color space to delete specific databits. In YC_bC_r, if we sort the components in descending order by eyes sensitivity, the serial will appear like this: Y (Luminance), C_b (Blue Difference), C_r (Red Difference). About 64% of the cones are red-sensitive, and about 2% are blue-sensitive [75, 76]. Moreover, brightness is more sensitive than colors. Less sensitive data bits are less noticeable to our eyes. JPEG takes advantage of this by removing a lot of color data bits from an image. This technique is called Chroma Subsampling. Another key point of JPEG is, it removes some of the high-frequency data bits from the image. Our eye is more sensitive to low-frequency data. To some extent, our eyes can not differentiate if the high-frequency data bits are removed from an image. To determine to what extent we can remove data bits, we try to use DCT to detect the higher frequency and the lower frequency pixels. Later, we delete some of the data bits by using quantization. [77].

Consequently, JPEG divides a picture into chunks of 8×8 blocks. JPEG needs 8 x 8 blocks to perform DCT on them. Every block contains 64 coefficients and every coefficients has 3 components (Y, C_b, C_r). To start, We subtract -128 from each of the pixel components to center all the values to 0. We find new values for our 8 x 8 blocks. Afterwards, we use Fast DCT (Discrete Cosine Transform) on our 8×8 matrix.

In (3.1), we see the equation of DCT [73], [74]. DCT is an algorithm that uses mathematical terms of cosine waves to transform values. In 8×8 blocks, DCT makes 64 cosine waves and sums all of them up. Hence, the final cosine wave of the block has some impact from all the 64 coefficients.

After using DCT, we find our new transformed 64 values, and we now have one Direct Current (DC) coefficient and 63 other Alternative Current (AC) coefficients. DC coefficient is the first coefficient and the most important one. It represents the general intensity of the whole block. DC coefficient remains flat. Other 63 coefficients excluding DC coefficient are Alternative Coefficients(AC). AC coefficients have less impact on the image as it only changes the intensity. If we take a closer look at Fig. 3.4, we see, in DCT, from the DC coefficient as we go horizontally by moving right or vertically



	0	1	2	3	4	5	6	7
0	16	11	10	16	24	40	51	61
1	12	12	14	19	26	58	60	55
2	14	13	16	24	40	57	69	56
3	14	17	22	29	51	87	80	62
4	18	22	37	56	68	109	103	77
5	24	35	55	64	81	104	113	92
6	49	64	78	87	103	121	120	101
7	72	92	95	98	112	100	103	99

(a) DCT Frequency coefficients

	0	1	2	3	4	5	6	7
0	17	18	24	47	99	99	99	99
1	18	21	26	66	99	99	99	99
2	24	26	56	99	99	99	99	99
3	47	66	99	99	99	99	99	99
4	99	99	99	99	99	99	99	99
5	99	99	99	99	99	99	99	99
6	99	99	99	99	99	99	99	99
7	99	99	99	99	99	99	99	99

(b) Skipped Bits for Y and C_r component

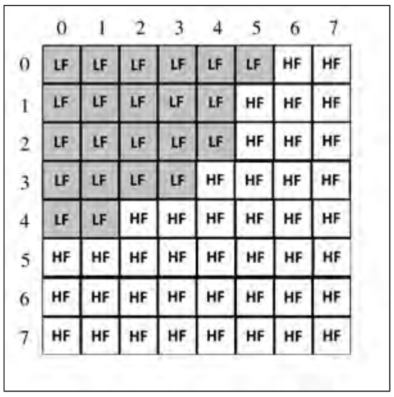
Figure 3.6: Separate quantization matrix tables for luminance 3.6a and chrominance 3.6b

by moving down to AC coefficients, the frequency keeps increasing.

We have our new 64 decimal values. Afterwards, we divide our values by the quantization matrix for quantization. DCT does not remove any data itself. It only identifies the lower frequency and higher frequency waves. Quantization is the process of removing the higher frequency data, which has less contribution to the image. There are separate quantization matrix tables for luminance and chrominance in Fig. (3.6a) and Fig. (3.6b). To find quantized DCT coefficients, we divide the values according to the quantization matrix tables. After dividing, we keep only the rounded values, delete the rest.

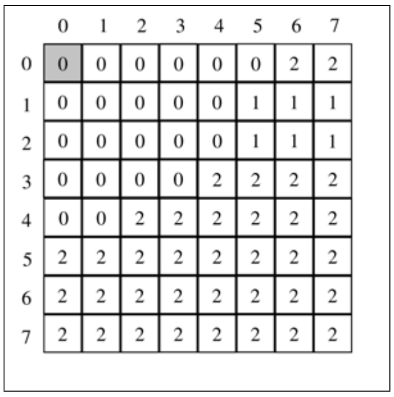
The data we just deleted is not renewable. Hence, it is a lossy process. We find the rounded values are mostly 0 or small numbers for high-frequency components [78]. We observe that the quantization matrix is smaller for those coefficients which are close to DC coefficients. Since we divide lower frequency components with small numbers, we get bigger (in a sense, it takes more bits to represent) values. Again, we divide the higher frequency components with larger numbers, resulting in smaller (in a sense, fewer bits to represent) numbers.

We modify our proposed Scan Script 4 (SS_4) and propose a lossy PJPEG architecture. First, We identify comparatively lower frequency coefficients. In Fig. 3.7a, we denote comparatively lower frequency pixels as LF, and comparatively higher frequency pixels as HF. We identify them by rigorously



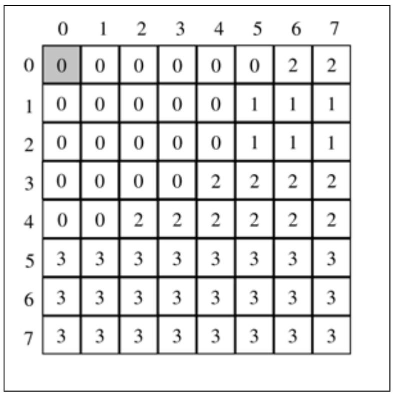
	0	1	2	3	4	5	6	7
0	LF	LF	LF	LF	LF	LF	HF	HF
1	LF	LF	LF	LF	LF	HF	HF	HF
2	LF	LF	LF	LF	LF	HF	HF	HF
3	LF	LF	LF	LF	HF	HF	HF	HF
4	LF	LF	HF	HF	HF	HF	HF	HF
5	HF	HF	HF	HF	HF	HF	HF	HF
6	HF	HF	HF	HF	HF	HF	HF	HF
7	HF	HF	HF	HF	HF	HF	HF	HF

(a) DCT Frequency coefficients



	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	2	2
1	0	0	0	0	0	1	1	1
2	0	0	0	0	0	1	1	1
3	0	0	0	0	2	2	2	2
4	0	0	2	2	2	2	2	2
5	2	2	2	2	2	2	2	2
6	2	2	2	2	2	2	2	2
7	2	2	2	2	2	2	2	2

(b) Skipped Bits for Y and C_r component



	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	2	2
1	0	0	0	0	0	1	1	1
2	0	0	0	0	0	1	1	1
3	0	0	0	0	2	2	2	2
4	0	0	2	2	2	2	2	2
5	3	3	3	3	3	3	3	3
6	3	3	3	3	3	3	3	3
7	3	3	3	3	3	3	3	3

(c) Skipped Bits for C_b component

Figure 3.7: In Fig. 3.7a identifies the lower frequency (LF) and higher frequency (HF) coefficients. $(1, 1)$ is the DC coefficient. Fig. 3.7b and Fig. 3.7c show the number of data bits we skip from each of the coefficients. Skipped bits are the same for component C_r and Y.

experimenting with the script. The pixels that have a huge impact on the image while skipping a data bit, we consider these as LF. Hence, we denote Coefficients $0-5$, $8-12$, $16-20$, $24-27$, $32-33$ as LF. Coefficients $6-7$, $13-15$, $21-23$, $28-31$, $34-63$ are HF.

We do not skip any bits for pixels $0-5$, $8-12$, $16-20$, $24-27$, and $32-33$. They have the highest impact on the image as it includes the DC and its closest AC coefficients. Later, We find $13-15$ and $21-23$; these pixels have a higher impact on the image compared to other AC coefficients. Hence, we skip only 1 bit from these pixels for all three components. For pixels $6-7, 28-31$ and $34-39$, we skip 2 bits for all of the components. C_b is the least sensitive color to our eyes. From $40-63$ pixels, we skip 3 bits for C_b. Only 2 bits for Y and C_r. Default Scan Script do not skip these bits and produce a larger image file size.

Fig. 3.7b and Fig. 3.7c show the number of data bits we skip each of the pixels. We skip 0 bits from 0th - 5th, 8th - 12th, 16th - 20th, 24th - 27th, 32nd - 33rd coefficients, 1 bit from 13th - 15th, 21st - 23rd coefficients, 2 bits from 6th - 7th, 28th - 31st, 34th - 39th coefficients for all the three components. Last, we skip 2 bits for Y and C_r, 3 bits for C_b from 40th - 63rd coefficients. We skip the bits from LSB. The bits we are skipping are deleted from the image.

Table 3.3: Cumulative size for five different images using default Scan Script (SS) [14]

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 1;	41K	346.75K	271.71K	1.19M	2.4M
0: 1-5, 0, 2;	128K	713.06K	657.11K	2.56M	5.4M
2: 1-63, 0, 1;	145K	836.36K	803.46K	2.92M	5.8M
1: 1-63, 0, 1;	163K	976.48K	947.92K	3.42M	6.3M
0: 6-63, 0, 2;	267K	1.08M	1.09M	4.03M	8.4M
0: 1-63, 2, 1;	406K	1.45M	1.60M	5.40M	14M
0,1,2: 0-0, 1, 0;	413K	1.51M	1.64M	5.67M	15M
2: 1-63, 1, 0 ;	433K	1.63M	1.76M	6.42M	15M
1: 1-63, 1, 0 ;	455K	1.75M	1.89M	7.15M	16M
0: 1-63, 1, 0 ;	636K	1.89M	2.77M	9.39M	25M

Table 3.4: Cumulative size for five images using SS_1

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 7;	9.1K	84.24K	64.83K	317.19K	734K
0,1,2: 0-0, 7, 6;	17K	144.15K	107.01K	597.58K	1.4M
0,1,2: 0-0, 6, 5;	24K	204.05K	148.32K	878.17K	2.1M
0,1,2: 0-0, 5, 4;	30K	263.82K	189.39K	1.13M	2.9M
0,1,2: 0-0, 4, 3;	37K	323.33K	230.44K	1.40M	3.6M
0,1,2: 0-0, 3, 2;	43K	382.75K	271.46K	1.67M	4.2M
0,1,2: 0-0, 2, 1;	50K	442.05K	312.49K	1.94M	4.9M
0,1,2: 0-0, 1, 0;	57K	501.34K	353.49K	2.21M	5.5M
0: 1-27, 0, 1 ;	381K	1.34M	1.42M	5.63M	16M
2: 1-27, 0, 1 ;	397K	1.46M	1.56M	5.99M	16M
1: 1-27, 0, 1 ;	415K	1.60M	1.70M	6.49M	17M
0: 28-63, 0, 1 ;	431K	1.60M	1.70M	6.49M	17M
2: 28-63, 0, 1 ;	431K	1.60M	1.70M	6.49M	17M
1: 28-63, 0, 1 ;	431K	1.60M	1.70M	6.49M	17M
0: 1-63, 1, 0 ;	612K	1.74M	2.58M	8.73M	26M
2: 1-63, 1, 0 ;	632K	1.86M	2.71M	9.48M	27M
1: 1-63, 1, 0 ;	654K	1.99M	2.84M	10.21M	28M

3.5 Performance Evaluation

We evaluate the performance of our proposed architectures through a real implementation.

Table 3.5: Cumulative size for five images using SS_2

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 6;	12K	105.72K	84.84K	362.15K	802K
0,1,2: 0-0, 6, 5;	19K	165.63K	126.15K	642.74K	1.5M
0,1,2: 0-0, 5, 4;	26K	225.40K	167.21K	923.05K	2.3M
0,1,2: 0-0, 4, 3;	32K	284.91K	208.27K	1.17M	3.0M
0,1,2: 0-0, 3, 2;	39K	344.32K	249.29K	1.44M	3.6M
0,1,2: 0-0, 2, 1;	46K	403.63K	290.32K	1.71M	4.3M
0,1,2: 0-0, 1, 0;	52K	462.91K	331.31K	1.98M	4.9M
0: 1-27, 0, 1;	377K	1.30M	1.39M	5.40M	15M
2: 1-27, 0, 1;	393K	1.42M	1.54M	5.76M	16M
1: 1-27, 0, 1;	411K	1.56M	1.68M	6.26M	16M
0: 28-63, 0, 1;	426K	1.56M	1.68M	6.26M	17M
2: 28-63, 0, 1;	427K	1.56M	1.68M	6.26M	17M
1: 28-63, 0, 1;	427K	1.56M	1.68M	6.26M	17M
0: 1-63, 1, 0;	607K	1.70M	2.56M	8.50M	26M
2: 1-63, 1, 0;	628K	1.83M	2.69M	9.25M	26M
1: 1-63, 1, 0;	650K	1.95M	2.82M	9.98M	27M

Table 3.6: Cumulative size for five images using SS_3

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 5;	17K	138.02K	115.44K	445.45K	921K
0,1,2: 0-0, 5, 4;	24K	197.79K	156.51K	725.76K	1.7M
0,1,2: 0-0, 4, 3;	30K	257.29K	197.56K	0.98M	2.4M
0,1,2: 0-0, 3, 2;	37K	316.71K	238.58K	1.25M	3.0M
0,1,2: 0-0, 2, 1;	43K	376.02	279.61K	1.52M	3.7M
0,1,2: 0-0, 1, 0;	50K	435.30K	320.61K	1.79M	4.4M
0: 1-27, 0, 1;	374K	1.28M	1.38M	5.21M	15M
2: 1-27, 0, 1;	391K	1.40M	1.53M	5.56M	15M
1: 1-27, 0, 1;	408K	1.53M	1.67M	6.07M	16M
0: 28-63, 0, 1;	424K	1.54M	1.67M	6.07M	16M
2: 28-63, 0, 1;	424K	1.54M	1.67M	6.07M	16M
1: 28-63, 0, 1;	425K	1.54M	1.67M	6.07M	16M
0: 1-63, 1, 0;	605K	1.68M	2.55M	8.30M	25M
2: 1-63, 1, 0;	625K	1.80M	2.68M	9.06M	26M
1: 1-63, 1, 0;	647K	1.92M	2.81M	9.79M	27M

Table 3.7: Cumulative size for five images using SS_4 (Proposed)

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 4;	22K	186.71K	150.79K	577.73K	1.1M
0,1,2: 0-0, 4, 3;	29K	246.22K	191.85K	857.32K	1.8M
0,1,2: 0-0, 3, 2;	36K	305.64K	232.87K	1.10M	2.5M
0,1,2: 0-0, 2, 1;	42K	364.94K	273.90K	1.37M	2.1M
0,1,2: 0-0, 1, 0;	49K	424.23K	314.89K	1.65M	3.8M
0: 1-27, 0, 1;	373K	1.27M	1.38M	5.06M	14M
2: 1-27, 0, 1;	389K	1.39M	1.52M	5.42M	15M
1: 1-27, 0, 1;	407K	1.52M	1.66M	5.92M	15M
0: 28-63, 0, 1;	423K	1.52M	1.66M	5.92M	15M
2: 28-63, 0, 1;	423K	1.52M	1.66M	5.92M	15M
1: 28-63, 0, 1;	423K	1.52M	1.66M	5.92M	15M
0: 1-63, 1, 0;	604K	1.67M	2.54M	8.16M	25M
2: 1-63, 1, 0;	624K	1.79M	2.67M	8.91M	25M
1: 1-63, 1, 0;	646K	1.91M	2.80M	9.64M	26M

Table 3.8: Cumulative size for five images using SS_5

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 3;	28K	238.35K	190.20K	743.16K	1.4M
0,1,2: 0-0, 3, 2;	35K	297.77K	231.22K	0.99M	2.1M
0,1,2: 0-0, 2, 1;	42K	357.07K	272.25K	1.26M	2.8M
0,1,2: 0-0, 1, 0;	48K	416.36K	313.24K	1.53M	3.4M
0: 1-27, 0, 1;	373K	1.26M	1.38M	4.95M	14M
2: 1-27, 0, 1;	389K	1.38M	1.52M	5.31M	14M
1: 1-27, 0, 1;	407K	1.52M	1.66M	5.81M	15M
0: 28-63, 0, 1;	422K	1.52M	1.66M	5.81M	15M
2: 28-63, 0, 1;	423K	1.52M	1.66M	5.81M	15M
1: 28-63, 0, 1;	423K	1.52M	1.66M	5.81M	15M
0: 1-63, 1, 0;	603K	1.66M	2.54M	8.05M	24M
2: 1-63, 1, 0;	624K	1.78M	2.67M	8.80M	25M
1: 1-63, 1, 0;	646K	1.91M	2.80M	9.53M	26M

3.5.1 Experimental Testbed Setup

We use real high-resource machines for deploying testbed servers in Canada. We create these servers using virtual machines, hosted in a physical data center. Here, we use two proxy servers, three account-container servers, three object servers for the media storage cluster. We use AMD Opteron 62xx class CPU, and OS Cent-OS 7. The memory and disk configurations of our Swift servers here cover- 1) two

Table 3.9: Cumulative size for five images using SS_6

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 2;	35K	294.39K	231.18K	0.94M	1.9M
0,1,2: 0-0, 2, 1;	41K	353.70K	272.21K	1.21M	2.5M
0,1,2: 0-0, 1, 0;	48K	412.98K	313.20K	1.48M	3.2M
0: 1-27, 0, 1;	372K	1.25M	1.38M	4.89M	14M
2: 1-27, 0, 1;	389K	1.38M	1.52M	5.25M	14M
1: 1-27, 0, 1;	406K	1.51M	1.66M	5.75M	14M
0: 28-63, 0, 1;	422K	1.51M	1.66M	5.75M	15M
2: 28-63, 0, 1;	422K	1.51M	1.66M	5.75M	15M
1: 28-63, 0, 1;	423K	1.51M	1.66M	5.75M	15M
0: 1-63, 1, 0;	603K	1.66M	2.54M	7.99M	24M
2: 1-63, 1, 0;	623K	1.78M	2.67M	8.75M	25M
1: 1-63, 1, 0;	645K	1.90M	2.80M	9.47M	26M

Table 3.10: Cumulative size for five images using SS_7

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 1;	41K	346.75K	271.71K	1.19M	2.4M
0,1,2: 0-0, 1, 0;	48K	406.03K	312.71K	1.46M	3.1M
0: 1-27, 0, 1;	372K	1.25M	1.38M	4.88M	13M
2: 1-27, 0, 1;	389K	1.37M	1.52M	5.23M	14M
1: 1-27, 0, 1;	406K	1.51M	1.66M	5.73M	14M
0: 28-63, 0, 1;	422K	1.51M	1.66M	5.73M	15M
2: 28-63, 0, 1;	422K	1.51M	1.66M	5.73M	15M
1: 28-63, 0, 1;	422K	1.51M	1.66M	5.73M	15M
0: 1-63, 1, 0;	603K	1.65M	2.54M	7.97M	24M
2: 1-63, 1, 0;	623K	1.77M	2.67M	8.73M	25M
1: 1-63, 1, 0;	645K	1.90M	2.80M	9.45M	25M

proxies each having one 8 GB memory and one 20 GB disk. 2) three account-containers each having one 8 GB memory and three disks each of 50 GB. 3) three objects having one 8 GB memory and three disks each of 700 GB. Each server has six 1 GB network interface cards. Fig. 3.8 and Table 3.13 present the experimental setup of our testbed. In addition, we deploy a private media cloud Secure Processing-aware Media Storage (SPMS) using OpenStack Swift (stable newton branch) with three replicas ($r = 3$) and 16384 partitions ($p = 16384$). There are nine devices for the account, container, and object ring files. Hence, each device has around 5461 partitions in $/srv/node/ < server >$ folders (devices are mount in this location according to OpenStack Swift guide [3]). Moreover, we implement

Table 3.11: Cumulative size for five images using SS_8

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 0;	48K	397.61K	312.96K	1.45M	3.0M
0: 1-27, 0, 1;	372K	1.24M	1.38M	4.87M	13M
2: 1-27, 0, 1;	389K	1.36M	1.52M	5.23M	14M
1: 1-27, 0, 1;	406K	1.50M	1.66M	5.73M	14M
0: 28-63, 0, 1;	422K	1.50M	1.66M	5.73M	15M
2: 28-63, 0, 1;	422K	1.50M	1.66M	5.73M	15M
1: 28-63, 0, 1;	423K	1.50M	1.66M	5.73M	15M
0: 1-63, 1, 0;	603K	1.64M	2.54M	7.96M	24M
2: 1-63, 1, 0;	623K	1.76M	2.67M	8.72M	24M
1: 1-63, 1, 0;	645K	1.89M	2.80M	9.45M	25M

Table 3.12: MOS and SSIM values of four images for the first Scan of all the 8 Scan Scripts. First Scan for different eight combinations are denoted as Scan1 - Scan8. MOS is calculated using 25 observers and SSIM is calculated using the VQMT tool [15]

First Scan	Avg. MOS	SSIM			
	Images	Image1	Image2	Image3	Image4
Scan1	0.51	0.29	0.62	0.60	0.54
Scan2	0.58	0.38	0.69	0.65	0.60
Scan3	0.65	0.42	0.82	0.69	0.65
Scan4	0.67	0.44	0.85	0.69	0.69
Scan5	0.68	0.45	0.87	0.70	0.70
Scan6	0.68	0.45	0.87	0.70	0.70
Scan7	0.68	0.45	0.87	0.70	0.70
Scan8	0.68	0.45	0.87	0.70	0.70

a social site for both mobile and web users. The mobile site contains different features for social interactions such as free video calls, chats, feeds, stickers, and so on. The site has already experienced more than 5 million downloads. The images that are saved and processed on this site leverage the architectures we propose in this Chapter.

In this setup, we upload different types of data from clients to the development server for around eight months[2]. Besides, we create $10,000$ accounts and $10,000$ containers in the Swift cluster. We upload around 1M images and video files in those accounts. Hence, the number of objects (n) is 1M for our test-bed server. We upload around 1.5TB data. Therefore, total data becomes 1.5TB$\times$3 = 4.5TB in our development server.

[2]The users have uploaded objects (images) according to their personal preferences and choice in their real usages. Thus, all the objects are mostly different as they come from real usages. We choose these images as they represent the

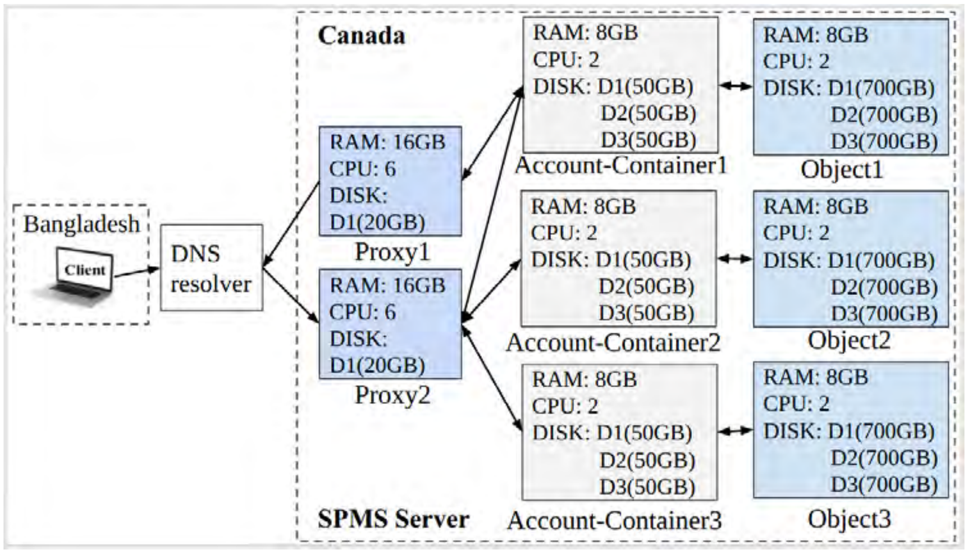


Figure 3.8: Testbed setup comprising a server in Canada and a client in Bangladesh

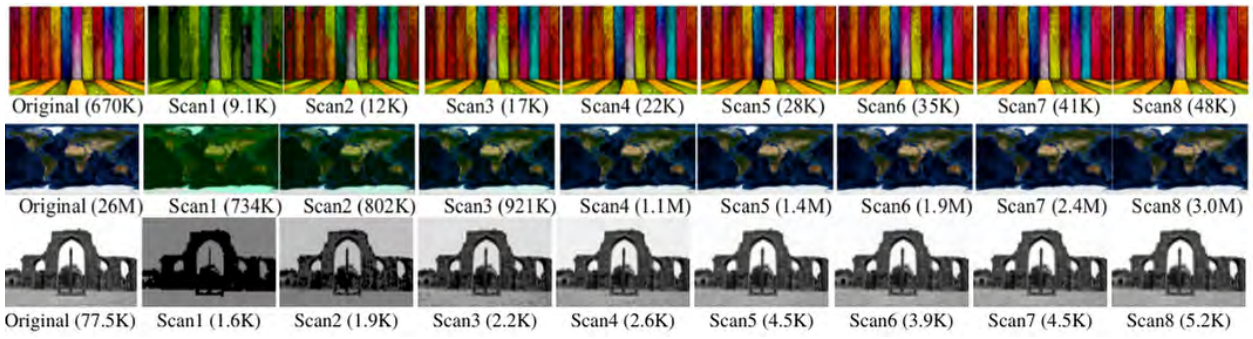


Figure 3.9: Comparison of first scan images for eight combinations. Here, Scan1 is (SS_{s1} of SS_1), Scan2 is (SS_{s1} of SS_2, Scan3 is (SS_{s1} of SS_3), Scan4 is (SS_{s1} of SS_4), Scan5 is (SS_{s1} of SS_5), Scan6 is (SS_{s1} of SS_6), Scan7 is (SS_{s1} of SS_7), and Scan8 is (SS_{s1} of SS_8).

Moreover, we use another web hosting server (Fig. 3.10) for a different purpose. We use this server to test a real case scenario for the difference between the load time of a normal image and our proposed algorithms. This server is located in London, UK. The client is located in Dhaka, Bangladesh. It has 30 hops from the client to London through hopping over Kansas, USA. Additionally, it hops to Kansas, United States, and then to London, UK. Note that, the performance will be affected depending on the distance between the locations of the client and the server. The loading time will increase by some milliseconds if the distance gets increased and vice versa. It will exhibit a similar effect in the case of the hop distance, i.e., the loading time will increase if the number of hops increases. To explore the

real-life testing of our proposed architecture.

Table 3.13: Configuration of machines used in testbed setup

Informations	Proxy Server	Object Server	Account-container Server	Client Machine
Architecture	x86_64	x86_64	x86_64	x86_64
CPU(s)	16	48	16	1
On-line CPU(s) list	0-15	0-47	0-15	0
Thread(s) per core	2	1	2	1
Core(s) per socket	4	12	4	1
Socket(s)	2	4	2	1
NUMA node(s)	2	8	2	1
CPU family	6	16	6	6
Model name	Intel(R) Xeon(R) CPU E5620 @2.40GHz	AMD Opteron-(tm) Processor 6174	Intel(R) Xeon(R) CPU E5620 @2.40GHz	QEMU Virtual CPU version 1.5.3
CPU MHz	2394.141	2199.967	2394.103	2393.998
Virtualization Type	VT-x	AMD-V	VT-x	Full Storage

impact, we change the location of the server to Singapore minimizing the number of hops from 30 to 11 while keeping the client in Dhaka, Bangladesh. After minimizing the number of hops, we observe a change of up to 25% difference in the loading time.

We create a custom dataset of 1333 pictures. Our selected dataset includes different sizes, resolutions, colorful, black and white images. We collect these pictures from datasets published in Kaggle [79], [80]. Table 3.14 shows the number of pictures of different sizes in the dataset.

To further evaluate our architectures, we use the MSCOCO2015 Test Dataset [81], which contains almost $81,000$ images of various categories.

Furthermore, we use a local virtual machine (Cent-OS 7) to calculate the cumulative size of our proposed scan scripts. We install libjpeg, libjpeg-turbo, and libjpeg-turbo-utils in the virtual machine [43]. We use thousands of images of different sizes for testing our proposed Scan Scripts.

We use Structural Similarity Index (SSIM) to calculate image quality to perform an objective-based evaluation using QoE [47]. For calculating SSIM, we use VQMT software [15] and method available

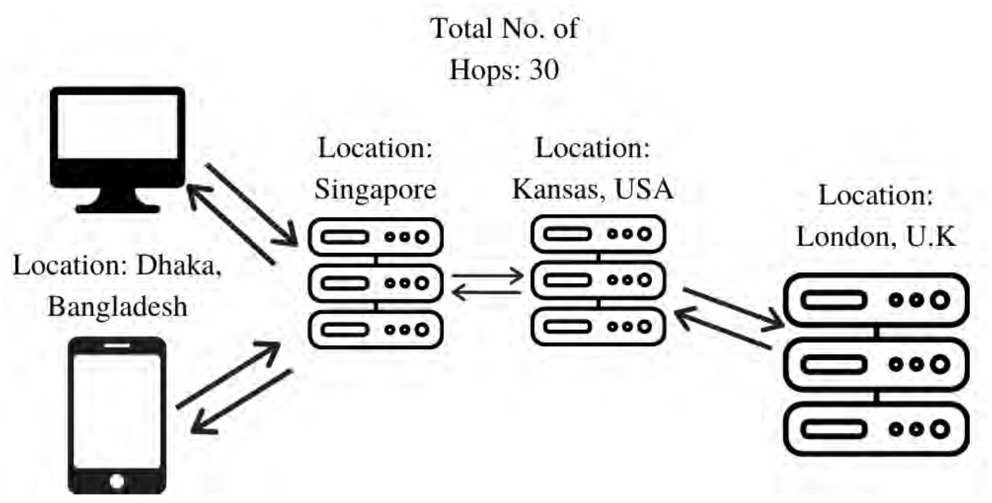


Figure 3.10: Testbed server setup to obtain performance of diversified remote devices using the web hosting server

to calculate SSIM in Scikit-Learn library in Python [82]. A higher SSIM value means more similar to the original image. Also, we use Python script to find the difference in file size between the original and compressed image.

3.5.2 Experimental Results

We describe the Experimental Results by our contributions separately.

3.5.2.1 Faster Image Retrieval

Table 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10, and 3.11 present the cumulative size of each Scan files using default Scan Script and Scan Script $1-8$ for five images. For our benchmarking process, each table contains the combinations of scanning images while converting them from baseline to progressive. Furthermore, we upload them into our cloud. Later, comparing their sizes after each phase of the Scans.

In Fig. 3.9, we present three images (Image1 of 670 KB, Image5 of 26 MB, and Image6 of 77.5KB) implementing the First Scans(SS_{S1}) for 8 Scan Scripts (SS_1-SS_8). We find, Scan4 to Scan8 all the images look exactly the same. As we use fewer bits for Scan4, we choose Scan Script 4 to compare with the default Scan Script. Scan7 encodes 7 DC coefficient bits same as default Scan Script in the

Table 3.14: PJPEG lossy architecture's results for the custom dataset

Size (kb)	1 -30	30 -100	100 -500	500 -1000	1000 -3000	3000 -8500
Pictures	198	419	467	72	81	96
Reduced %	10.49	20.84	23.26	24.93	21.72	25.31
SSIM	0.96	0.96	0.96	0.97	0.98	0.98

First Scan. Hence, we refer to Scan7 as the First Scan for default Scan Script.

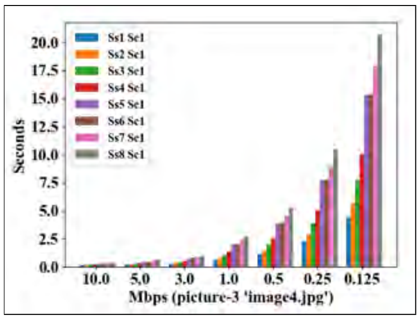
For subjective-based evaluation, we use Mean Opinion Score (MOS) [46, 47] metric. We request 25 observers to differentiate among the images of Fig. 3.9 to perform a subjective evaluation. All of them confirm that visual quality (V_q) is the same for the First Scan of Scan Script 4 (SS_4) and Scan Script 7 (SS_7). For objective based evaluation, Table 3.12 shows the MOS and SSIM values of four images for the First Scans of 8 Scan Scripts. In Table 3.12, We see the SSIM values are almost the same for Scan4 and Scan7.

Furthermore, we test First Scan of (SS_4) and (SS_7) in MSCOCO2015 Dataset. We find the average SSIM is 0.551 and 0.553 respectively. That verifies we get the same quality images for the First Scan's of Scan Script 4 and default Scan Script.

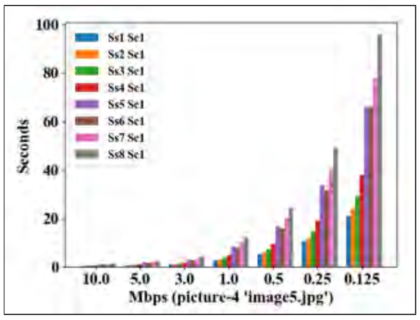
Lastly, we compare the load time of the actual picture and the picture generated by our proposed script in various State of The Art Cloud Applications such as Google Drive and Dropbox. Table 3.15 shows the load time difference between the pictures using the network section of Chrome DevTools [83]. We use 3 images (Image3, Image4, and Image5) to load our proposed First Scan of Scan Script 4. We load the images in different bandwidths. For example, Image3 in 0.125MBps, the original image loads in 66 seconds. Moreover, the image using our proposed Script takes only 34.06 seconds to load. It shows 48.39% improvement in our proposed Script. Table 3.15 confirms our images load faster on state-of-the-Art cloud applications as well.

3.5.2.2 Lossy PJPEG Architecture

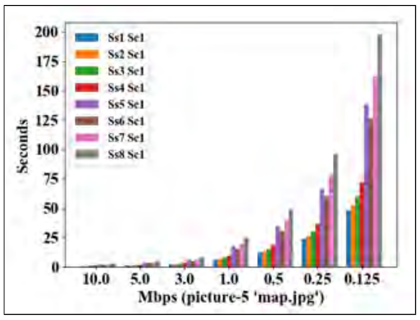
In Table 3.14, we test our PJPEG Lossy Architecture using our custom dataset. We see 198 images are within the range of file size $1 - 30$kb. For these small-sized images, the average SSIM result is 0.96. On average, the image file size is reduced up to 10%. A slightly larger image produces a greater compression result. In the last group, 96 images within the file size of 3MB to 8MB produce the



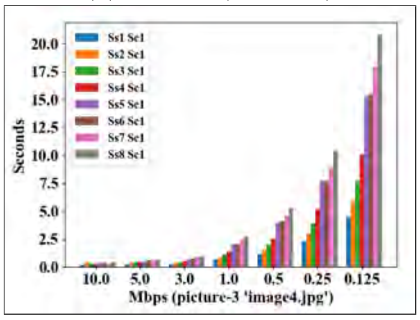
(a) Image3 (Desktop)



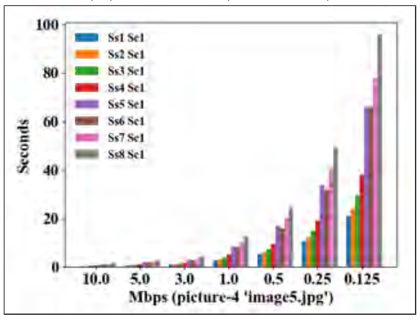
(b) Image4 (Desktop)



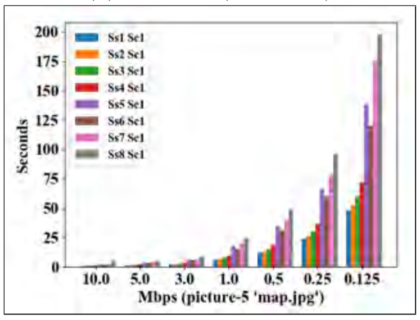
(c) Image5 (Desktop)



(d) Image3 (Mobile)



(e) Image4 (Mobile)



(f) Image5 (Mobile)

Figure 3.11: Time needed to load different Scan Scripts under different bandwidth availability (Ss = Scan Script, Sc = Scan number)

highest SSIM value of 0.98. It also produces the highest compression rate of reducing 25.31 percent more than regular JPEG standard compression.

Furthermore, we test our compression algorithm in the MSCOCO2015 dataset [81]. Our compression offers a 27.40% of reduction in file size than standard JPEG. The average SSIM result is 0.952.

In the first group, we have 198 images from $1 - 30$ kb. For these small-size images, the Average SSIM result is 0.96. On average, the image file size is reduced up to 10%. For images sized $30 - 100$ kb, we have 419 images. This group has an average SSIM of 0.96, and the file size reduction percentage is 20.84. A slightly larger image produce a greater compression result. This pattern continues for other groups too. In the last group, 96 images of 3 MB to 8 MB produce the highest SSIM values of 0.98. On average, it gives an SSIM value of 0.968. It also produces the highest compression rate of reducing 25.31 percent more than regular JPEG compression.

3.5.2.3 System Resource Usage

We explore system resource usages by our proposed solutions and the default mechanism. As per our exploration, both our proposed solutions and the default mechanism consume nearly the same amount of resources. To be specific, as measured by System Monitor, memory usage is almost 100 MB and CPU usage is close to $10 - 15\%$ in both cases.

When it comes to resource consumption, both architectures need nearly 0.1 GB of memory on average in both the cases and the CPU usage hasn't changed considerably since all work is performed in memory.

Table 3.15: Load time comparison between the actual picture and the picture generated by our proposed faster image retrieval scan script in state of she art cloud applications. Here, we counted the load time of only the image, not the UI

Image3	Loading Time and Its Improvement in Google Drive			Loading Time and Its Improvement in Dropbox		
Speed	Original Image (s)	Image using proposed Script (s)	Improvement in Proposed Script (%)	Original Image (s)	Image using proposed Script (s)	Improvement in Proposed Script (%)
10Mbps	1.17	0.832	28.89	1.86	1.73	6.99
5Mbps	1.83	1.18	35.52	2.62	1.79	31.68
3Mbps	3.52	1.36	61.36	6.21	5.66	8.86
1Mbps	10.05	5.99	40.40	19.79	17.37	12.23
0.5Mbps	17.44	9.43	45.93	28	18.76	33
0.25Mbps	34.29	17.16	49.95	72	72	0
0.125Mbps	66	34.06	48.39	114	96	15.79
Image4	Loading Time and Its Improvement in Google Drive			Loading Time and Its Improvement in Dropbox		
Speed	Original Image (s)	Image using proposed Script (s)	Improvement in Proposed Script (%)	Original Image (s)	Image using proposed Script (s)	Improvement in Proposed Script (%)
10Mbps	1.95	0.928	52.41	2.45	1.91	22.04
5Mbps	4.16	1.3	68.75	3.73	3.56	4.56
3Mbps	2.92	2.48	15.07	6.34	6.04	4.73
1Mbps	9.61	8.55	11.03	19.78	19.07	3.59
0.5Mbps	15.91	12.15	23.63	39.11	38.25	2.20
0.25Mbps	54.06	33.56	37.92	66	57.53	12.83
0.125Mbps	114	66	42.11	150	144	4
Image5	Loading Time and Its Improvement in Google Drive			Loading Time and Its Improvement in Dropbox		
Speed	Original Image (s)	Image using proposed Script (s)	Improvement in Proposed Script (%)	Original Image (s)	Image using proposed Script (s)	Improvement in Proposed Script (%)
10Mbps	**No Preview Available due to a big Resolution size of 21600 x 10800**		**N/A**	1.6	1.57	1.88
5Mbps				3.01	2.99	0.66
3Mbps				5.08	5.07	0.197
1Mbps				16.69	16.67	0.12
0.5Mbps				33.68	32.52	3.44
0.25Mbps				66	66	0
0.125Mbps				138	138	0

3.5.3 Experimental Findings

Findings are discussed separately for both of our contributions as before.

Table 3.16: Comparison of our proposed approach with other existing research studies

Name	Progre-ssive loading	Private cloud	Retrieving image faster	Efficient image storage	User waiting time	Underlying technology
Noor et al., [59]	✓	✓	✓	✓		Bicubic interpolation in iBuck
Yan et al., [72]	✓			✓		Dynamic Resizing
Abuzaher et al., [84]				✓		RGB Percentage Replacement
Hussain et al., [70]				✓		Modified Quantization and Arithmetic Encoding
Louie et al., [85]	✓		✓	✓		Segmented Compression and Transmission
Iqbal et al., [86]				✓		Modified Entropy Encoding
Mali et al., [87]				✓		Sparse RNN Smoothing and Learned Quantization
Lee et al., [88]	✓		✓			Trit-Planes Algorithm
Cai et al., [89]	✓				✓	CNN Based Progressive Image Compression Framework
Byju et al., [20]	✓		✓			Coarse Resolution and Wavelet Features
Lu et al., [90]	✓			✓		PLONQ with Nested Quantization
Abdollahi et al., [91]	✓					Recursive Least Squares(RLS) Adaptive Algorithm
Our Proposed Approach	✓	✓	✓	✓	✓	**Encoding Less bits in the First Scan**

3.5.3.1 Faster Image Retrieval

We approach to balance the trade-off between the number of Scans and the size of the images in the first scan. We focus to ensure that viewers get an optimum view after the first partial image loading.

Table 3.17: Quantitative comparison over improvement in performance achieved by our proposed approach and other existing research studies along with corresponding datasets under experimentation as reported in respective studies (CR refers to Compression Rate and BPP refers to Bits Per Pixel)

Name	Transmission time efficiency	Image quality	Storage efficiency	User waiting time	Dataset
Noor et al., [59]	Upto 25%	SSIM: 0.9113	By 31.75%	-	3 Datasets; [92], [93] and a Custom Dataset
Yan et al., [72]	-	PSNR: 32 dB	By 41%	-	MIR Flickr Dataset [94]
Abuzaher et al., [84]	-	-	By 55%	-	Not Mentioned
Hussain et al., [70]	-	PSNR: 38.9 dB	CR = 6.202 : 1	-	Custom Dataset
Louie et al., [85]	Upto 50% .	-	By 50%	-	Not Mentioned
Iqbal et al., [86]	-	SSIM: 0.999	1 BPP	-	Air Jet Image from JPEG AI Dataset [95]
Mali et al., [87]	-	SSIM: 0.8413	0.371 BPP	-	Kodak Dataset [96], Div2K [97]
Lee et al., [88]	-	PSNR: 35 dB	0.75 BPP	-	Kodak Dataset (For Verification) [96], and Vimeo90k Dataset [98]
Cai et al., [89]	-	PSNR: 40 dB	1.72 BPP	26% More than JPEG	Kodak Dataset [96]
Byju et al., [20]	Decoding Time 127.56s	-	-	-	Big Earth Dataset [99]
Lu et al., [90]	-	PSNR: 39 dB	1.5 BPP	-	JPEG AI Testset [95]
Abdollahi et al., [91]	-	PSNR: 21.7 dB	CR = 76 : 1	-	Custom Dataset
Our Proposed Approach	**Upto 69%**	**SSIM: 0.952**	**Upto 27%**	**54% Less than JPEG**	**MSCOCO2015 Dataset [81] and Custom Dataset**

Moreover, we ensure viewers do not wait for a long time to get the full image view because of a higher number of Scans. Considering these, after going through a rigorous bench-marking with some 50 images on our testbed, we observe that Scan Script 4 to Scan Script (SS_8) produce the same quality images in the First Scan. From them, Table 3.7 has a considerably fewer number of bits in the First Scan to generate a balanced view.

That means they are of the same quality. 25 observers also confirm that visual quality (V_q) is the same for the First Scan of Scan Script 4 (SS_4) and Scan Script 7 (SS_7).

Fig. 3.11 shows the improvement of time and size of the candidate images in our test-bed with our proposed Scan Script, compared with the default Scan Script. Our proposed Scan Script gains over 50% improvement (54% to be exact) considering the time it takes for the first view of a progressive image to satisfy a viewer with an optimum view. Besides, for remote and local VM servers, the network hop is 16 and 2, respectively. Moreover, the average incoming and outgoing network speeds in a client machine is 400 Bit/s where Ttl is 43.61 MByte.

However, while using MSCOCO2015 Dataset, 73 out of $81,000$ images are showing errors. These 73 images are black and white, and very small in size. The error does not occur for slightly larger-sized images. Later, we find that the default Scan Script also can not load these 73 images as well. We fix the error in our proposed script by removing chrominance components.

However, after loading all the scans, the image size is slightly larger for our proposed Scan Script 4. That is a minor drawback for our proposed Script.

3.5.3.2 Lossy PJPEG Architecture

While modifying our proposed Scan Script 4 (SS_4) to make a lossy architecture, we discover something unusual. we can not encode 32^{nd} pixel alone. To solve this we had to encode 32^{nd} and 33^{rd} pixel together, despite the fact that 33^{rd} pixel should be in the HF section. However, for making our scripts, we put 33^{rd} pixel in LF.

To make the lossy compression, we try making many Scan Scripts. At first, we make a script that can reduce the file size up to 40% without even compromising image quality. However, it makes images a bit blurry while compressing a smaller image file size. The script produces good quality images for greater than 700kb file size. The median size for images user usually consume is 200 to 2200kb on the Internet [8]. Hence, the script can not handle small size images. Therefore, we move forward to make another script that can maintain good quality for smaller images too. We come to know, the higher the image size is, the more we can delete data bits. Additionally, the more we delete data bits, the worse the image's quality becomes. Hence, we try removing fewer data bits to ensure the image quality. After experimenting more, we make a lossy PJPEG scan script that works for the smaller image file size. To use our lossy PJPEG architecture with having great results, we need a minimum image size of at least 6kb. Most used pictures on the Internet are greater than 6kb. Hence, it is not something that we should worry about. The bigger the image file size, the better SSIM we get, and the more we can reduce the image size. In our result Table 3.14, we see that our compression approach works better with larger images.

3.5.4 Comparison of Our Approach with Other Studies

As we have discussed earlier, progressive JPEG offers advantages under environments where bandwidth is a big factor of constraint. We compare our proposed approach with other recent existing research studies in Table 3.16 and 3.17. These tables compare the studies in qualitative and quantitative manners respectively. Here, Table 3.16 presents a qualitative comparison and Table 3.17 presents a quantitative comparison among the studies under. As shown in the Table 3.16, existing progressive JPEG based related research studies [20, 72, 85, 88–91] use different technologies such as Dynamic Resizing, Segmented Compression, Trit-Planes Algorithm, Progressive Latent Ordering Nested Quantization (PLONQ), etc., to reduce file size for the overall image. Reducing file size leads to less retrieval time and transmission time for the full quality image. However, most of the existing studies do not focus on the notion of faster image preview even though faster image preview decreases user waiting time. A research study [89] significantly improves first preview time from JPEG2000 [100], BPG [101], Balle [102], WebP [103], and Toderici [104]. However, this study do not perform better than JPEG [73] and eventually have ended up with 26% increased user waiting time for JPEG. On the contrary, our proposed approach decreases user waiting time for JPEG by 54%. Here, our proposed approach adopts a new Scan Script for performing the first scan in road to ensuring a faster image preview. This, in turn, results in a faster loading of images using our proposed approach compared to other existing technologies.

Besides, storing images in public clouds can significantly increase retrieval delay. Using private clouds for managing images could present a remedy here, which is sparsely focused in the literature. In this regard, our previous study [59] works on a framework for secured image processing in a private cloud. Following our previous study, this Chapter attempts to fill up the gap in the literature by using a private cloud for the purposes of faster loading and retrieving images along with managing the storage efficiently. Thus, in summary, this Chapter realizes the notion of first scan to enable progressive loading and manages images over a private cloud, which in combination result in faster image retrieval as well as efficient image storage. Such a combination is new in the literature to the best of our knowledge as shown in Table 3.16 when positioned against state-of-the-art. Nonetheless, Table 3.17 demonstrates that our proposed approach mostly works better than all other state-of-the art approaches in terms of transmission efficiency, image quality, storage efficiency, and user waiting time in combination.

3.6 Conclusion and Future Work

In this Chapter, we investigate an important problem in the realm of cloud-related image communication and storage from the perspective of its efficient retrieval. In this regard, we point to a significant gap in the literature on efficient retrieval and storage of progressive images - especially in bandwidth-constrained cases. Accordingly, we propose an orchestration methodology through a new image scanning technique and a new lossy compression technique. We implement the proposed orchestration in a real setup over two different continents, comprising a server in Canada and a client in Bangladesh enabling a private cloud architecture. We conduct rigorous experimentation to perform both system-level and subjective evaluations over the experimental setup. The evaluation results confirm that we can achieve substantial performance improvement using our proposed orchestration. Our future work includes system-level exploration of the next-generation JPEG images to improve image storage quality further.

Part II: Device-sensitive Multimedia Uploading, Retrieval, Searching, and Archival

Chapter 4

Secure Processing-aware Media Storage and Archival System (SPMSA)

4.1 Introduction

With the rapid growth of embedded devices, media industries have started facing challenges in storing, processing, and managing large amount of data. The data include video, photos, audio, text, etc. Users are producing and consuming such data more than ever with social media, online video, user-uploaded contents, gaming, Software-as-a-Service applications, etc., [3]. All these applications present a common need for easily-accessible storage systems that can potentially grow without bounds or limits.

Besides, with the advancement of technology, one important sector that is experiencing rapid growth in its field nowadays is media the archival system. As a part of conventional methods, many aspects such as CCTV usage is perhaps at its peak in recent times [23]. In most cases, CCTV videos are stored locally and demand a huge amount of storage space as the stored uncompressed video data is of substantial size. Moreover, the data is only stored at the local hard drive, which is vulnerable to physical damage, hardware failure, firmware corruption, etc. Overcoming these vulnerabilities demands a system that can handle large video files easily and keep them safe. In this regard, cloud storage has often become a necessity for the purpose of ensuring high availability of video data, as one might need to access the video data from a remote place using diversified devices.

4.1.1 Existing Studies on Media Storage and Archival Systems

Conventional services such as Dropbox, Sync, SugarSync, Live drive, Google drive, etc., provide popular storage systems for storing all types of files including media files [105–110]. These public storage service providers are popular for their file syncing features. The offered services appear mostly to be a black box to the users, where the users can put files without having any idea of background processing. However, extra processing is needed in the middle for the services. Examples of extra processing of media server include the tasks of media processing to retrieve data from storage to the end devices. Imemories [111], Cloudinary [112], etc. provide media cloud to perform such tasks of media processing on the fly, which are generally costly for the media servers. While performing the tasks of media processing, some cloud providers offer higher security without giving any type of conversion of files, whereas others provide facilities to process video without guaranteeing user privacy [105].

Recent research studies investigate CCTV surveillance systems with cloud storage servers [22, 23, 113]. A study in this regard [114] proposes a three-layer cloud-based video monitoring and analysis system that investigates several aspects such as processing time, human monitoring, GPU space and seeks to find a solution. But the entire process is very GPU intensive and their system was tested with low-resolution videos (704×528). Another study [115] , the proposed method includes exploring large-scale video retrieval using a layered architecture and deep learning being augmented with semantic approaches. named IntelliBVR. Though the processing power is very high to make it available to everyone with the minimal financial cost and its results are not generated as fast as expected. However, none of these existing research studies investigates efficient storing and long-term archiving of the video data.

4.1.2 Motivations and Challenges

Conventional cloud-storage services enforce security mostly on public-facing data transfers using SSL/TLS of HTTPS [116]. The usage authenticates web servers and encrypts messages sent between browsers and web servers [117]. On top of that, in SPMSA, our method is to enforce security on internal-traffic between proxy and storage nodes while syncing between different regions and on-disk data. Such enforcement escalate the level of overall security specially for the needs that are owned and managed by different entities.

Existing literature is yet to focus on an important realm of cloud-based video surveillance systems involving long-term storing and archiving [11, 53]. Here, handling a huge amount of video data on a daily basis and making it available to regular clients at any given time present a challenging problem to solve. This happens as CCTV produces a huge amount of video data on a regular basis, and data needs to be handled carefully. In the data handling task, the video data needs to go through some processing to be ready to store in a cloud server with ease. In addition to that, the availability and the process of storing and retrieving the data easily has the most priority. Hence, we need to focus on storing large video files efficiently and archive the data for long-term purposes in a better manner enabling more space utilization.

The focus has become more important for developing and underdeveloped countries, which often have limited Internet access and employ limited storage space. Accordingly, CCTV applications in such countries use direct storage systems that can only hold videos for a limited period of time [118]. Here, continuous access to public cloud-based systems is not viable as it demands large upfront costs, high bandwidth, and continuous internet access [119]. Thus, the conventional alternative solutions utilized in this regard experience the following challenges.

- Challenge-1: Conventional media data storing and archival systems use online pre-processing rather than offline pre-processing.
- Challenge-2: Storing media data on a local storage device creates a single point of failure and a high cost of data storage [118]. This approach also has the risk of physical damages and hardware failure.
- Challenge-3: Storing media directly to a cloud storage location addresses the problem of a single point of failure, however, requires large upfront costs, high bandwidth, and constant internet access [120].
- Challenge-4: Using application-specific solutions such as security-enabling direct facial recognition technology and event triggers an attempt to minimize the number of recordings, however, such systems cannot be 100% accurate [121].

4.1.3 Implications of Our Study

To address these challenges, we propose a new private cloud paradigm namely 'Secure Processing-aware Media Storage and Archival' (SPMSA), which unifies object storage (Storage-as-a-Service) with cloud security, media processing (Processing-as-a-Service) and archiving media. Here, our method is to design an in-storage media processing system along with performing encryption-decryption based on user demand. To do so we exploit middleware services available in OpenStack Swift, an open-source object storage platform [4]. Besides, we design a new proxy server for performing not only proxy related tasks but also tasks related to media processing, resizing, video transcoding, and encryption-decryption of media files. Hence, we name the proxy server as proxy-media server. The proxy-media server is scalable by nature and can easily be DNS-load-balanced. Here, both upload/download traffic can be separated and backed by each-other through properly designed domain-name (DNS-entry) [122]. Thus, the proxy-media server omits the need of conventional public cloud servers to use different media servers for processing media files. This saves substantial bandwidth and time of conventional public cloud spent in communicating with media servers.

In road to implementing our proposed SPMSA, we develop three new middleware services named 'PhotoPool', 'MediaBucket', and 'SecureCloud'. Here, PhotoPool middleware is mainly responsible for resizing and converting images to PJPEG. SecureCloud middleware performs encryption-decryption of the image files using PFCC algorithm [53]. Besides, MediaBucket middleware is responsible for video related processing such as transcoding. Combination of all these three middleware services make our proposed cloud architecture more secure, highly-scalable, and faster-accessible to end users for all types of media files. There are many diverse and sensitive applications for our proposed architecture. Examples include medical imaging systems, military media communication, mobile commerce, social media, etc. To achieve large volume of media data such a surveillance data we also propose an archival system for such activities.

Additionally, in this Chapter, we propose a new CCTV surveillance system that stores video data in multiple locations - at first in the local storage, where the CCTV data will be stored for a short period, and then cloud storage at an interval of the regular time period for better data availability and fault tolerance. Finally, in the archive storage, *compressed* data will be archived for a long time throughout the replication process. In this system, the video data will have high availability. Additionally, through a proposed intuitive user interface in our system, a client with no prior knowledge about security can easily access the video data. Nonetheless, as per real experimentation, our proposed system uses 55% less storage compared to SPMS [11] without demanding constant internet connectivity such as Google's NEST CAM [121]. Moreover, our proposed system is far more robust than conventional physical storage [118] and provides improved storing time when compared to popular solutions such as Google drive, Dropbox, and iCloud.

4.1.4 Our Contributions

We make the following set of contributions in this Chapter.

- We propose a new private cloud paradigm that enables media processing in parallel to enforcing security and we implement the proposed paradigm through developing three new middleware services to demonstrate its applicability in real systems.
- We propose an efficient media data archival system to store and archive large videos such as surveillance data through segmentation and compression by leveraging a private OpenStack Swift object service.
- We store multiple replicas of the same video data in two separate servers (storage server and archive server), thereby ensuring reliability and fault tolerance. We use video encoding to compress the video data and segment them before uploading them to servers.
- We perform rigorous experimentation to evaluate the performance of the proposed and developed system in a real testbed and compare its performance against conventional popular alternatives such as Dropbox, iCloud, and Google Drive.
- Experimental results demonstrate significant performance improvement using our proposed paradigm compared to the conventional one.

4.2 Background

In this Section, at first we discuss about OpenStack Swift an open source object storage system. After, we present the QoE metrics used for validating our experimental results.

4.2.1 OpenStack Swift

OpenStack Swift provides many services in this one software, among these computing services, storage services are mostly used to create a cloud computing service experience.

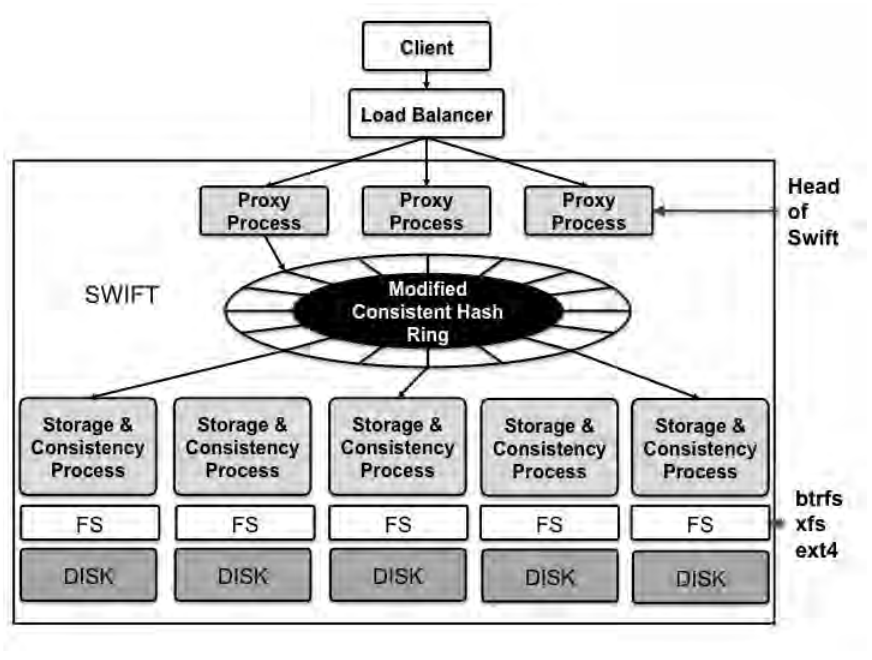

Client
Load Balancer
Proxy Process
Proxy Process
Proxy Process
Head of Swift
SWIFT
Modified Consistent Hash Ring
Storage & Consistency Process
Storage & Consistency Process
Storage & Consistency Process
Storage & Consistency Process
Storage & Consistency Process
FS
FS
FS
FS
FS
btrfs
xfs
ext4
DISK
DISK
DISK
DISK
DISK

Figure 4.1: Overview of OpenStack Swift architecture [11]

It is part of OpenStack and is completely free to use. Because of its efficiency and scalability, many major companies use Swift as their go-to cloud storage. Swift is meant to be run on Linux distributions and on any x86 hardware setup. Swift has an architecture called "Eventual Consistency Architecture". This allows Swift to create enormous cloud infrastructures, which can store tons of unstructured data. Objects in Object Storage have an unique identifier. In Swift, each object has an URL given to them. Objects can be accessed by going to the provided URL. Data can be stored and retrieved from the Swift server in a specific way. Globally popular RESTful HTTP API is the most popular way to do so. More on these is discussed in the Swift API section.

Figure 4.1 shows the architectural overview of SWIFT. The first step is understanding how a file is being saved on an object storage system. The object storage system is divided into several parts or components. A proxy server is located in the first layer. Data that goes in and out of the storage has to go through the HTTP file transfer protocol. The requests for data are done by API requests. The task of the proxy server is to capture the requests and work accordingly. The proxy server determines the location of the data or it's storage node by the URL. After this, there are Rings, which keep the address of the information like names and entries which are stored on the cluster and also keep track of the actual physical location of the data.

The way Rings keep the mapping work by introducing zones, devices, partitions, and replicas. Zones might be any storage device like a hard drive to a full server. After that, there are containers and accounts. The list of containers in a particular account is stored in that account's database [123].

4.2.2 QoE Measurements

To quantify user's QoE, both objective factors and subjective factors need to be considered. Objective factors include parameters in Nework layer (jitter, packet loss, delay, etc.) and Application layer (resolution, frame rate, etc.) [52]. Objective factors determine visual difference of a video from its definition, which is influenced by resolution and bit rate. Compared with the objective factors, subjective factors are more complex. The subjective factors can be extended to users psychological conditions such as preference and users profile information (age, gender, etc.). For subjective and objective measurements of QoE, commonly used video quality metrics are [46, 47] as follows:

MOS: According to ITU-T Recommendation P.910, MOS is chosen as the score criterion for subjective QoE measurement, which reflects the appraisal of some test panel. The MOS is expressed on a five-point scale, where 5 = excellent, 4 = good, 3 = fair, 2 = poor and 1 = bad. The minimum threshold for acceptable quality corresponds to a MOS of 3.5 [48].

PSNR: Peak-Signal-to-Noise-Ratio gives the ratio (in dB) between power of the original signal and power of a reconstructed compressed signal. PSNR is usually derived via mean squared error (MSE) between two signals in relation to the maximum possible luminance of images. MSE and PSNR are calculated as [46, 47]:

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2 \tag{4.1}$$

$$PSNR = 10\log_{10}\left(\frac{{MAX_I}^2}{MSE}\right) \tag{4.2}$$

Equation 4.1 and 4.2 present equations of MSE and PSNR of a noise-free $m \times n$ monochrome image I and its noisy approximation K. Here, MAX_I is the maximum possible pixel value of the image. Although PSNR may not always accurately reflect the QoE, as demonstrated in [49], it continues to be a popular method to evaluate quality difference among videos.

SSIM: Structural Similarity Index [50] uses a structural distortion based measurement approach. Structure and similarity in this context refer to samples of the signals having strong dependencies between each other, specially when they are close in space [51]. Here, the rationale is that human vision is specialized in extracting structural information from the viewing field, not in extracting errors.

VQM: Video Quality Metric [52] measures the perceptual effects of video impairments including blurring, jerky/unnatural motion, global noise, block distortion, color distortion, and combination of them.

In summary, higher MOS, higher PSNR, higher SSIM, and lower VQM describe videos as better one.

4.3 Related Work

Images and videos have considerable value for diversified purposes including sensitive applications such as medical diagnosis, military communication, etc. Security is becoming an utmost important issue in communication and storage of such files. A possible solution to this issue is using private image clouds [1, 12, 53], which can provide image security as well as faster availability of those images at any ends. Nonetheless, increasing quality factors such as high frame rates and ultra-high resolution videos raise the need for efficient and scalable computation. However, to the best of our knowledge, there is still no such efficient and scalable private cloud for both maintaining image data securely, transcoding video files, and archiving media data while making them highly available. Such clouds are necessary when a user wants to securely access important media files anywhere using any type of remote devices. Hence, resizing of images [53], transcoding of videos [11], and archival of video surveillance system come into play. If computations needed for such resizing and transcoding are done on the fly, then processing power and delay can severely increase [111, 112] undermining the overall goal. In below subsections, at first we present the existing studies on media storage clouds. After, we discuss on recent studies related to video surveillance system and media archival.

4.3.1 Literature on Media Storage Clouds

Best cloud-storage solutions available to date, e.g, Dropbox, Sync, SugarSync, Live drive, Google drive, etc., provide inexpensive benefits with their own limitations [105]. For example, Sync offers a robust, well-rounded service with fast transfer speed, end-to-end encryption using AES-256, and automatic backup features. However, the encryption slows down uploads and previews. Besides, mobile clients lack file sharing and limited syncing in this cloud often makes discomforts to maximum middle-sized companies [107]. Alternatively, users of SugarSync get access to up-to-date data on any device exploiting local encryption and TLS in transit. However, limitations of this cloud service include slow response time, lack of backup scheduling, etc., [108]. Besides, Livedrive offers web and client applications with extra features such as integrated media player, file sharing, etc. However, its hidden pricing and clunky user interface are the main obstacles in road to make it popular [109]. In addition, Dropbox [106] offers syncing, sharing, and third-party integrity with backup solutions. Google drives [110] also offer sharing and document collaborating features. However, all these cloud solutions provide only storage without providing any scalable media processing such as image processing or resizing, video transcoding, etc.

Cloudinary [112] provides a simple integration facility to users to upload, transcode, manipulate and deliver videos via a global Content Delivery Network (CDN). Besides, iMemories [111] turns movies and photos into a digital format so that they can easily be viewed and shared on different modern devices. These media cloud solutions perform media processing while delivering from the storage, not in cloud-storage. Furthermore, vendors can use such type of storage solutions through paying their own. But, if they want to build their own media cloud with special facility enhancing security through choosing best encryption algorithms, add more media features, and so on, then they should not depend on traditional storage systems. They have to find out which tools are good for resizing and transcoding of video files, what resolutions are needed for serving diversified client devices, how to encrypt and decrypt video files for faster outcome. These studies focus on many traditional cloud storage systems, but they are just service providers, not focus on how to build a secured media cloud from scratch. Recently, many social networking applications are emerging such as ringID, ringID Studio, etc. [124] using their own private cloud. Hence, there are still many vendors who want to make their own distributed system for media cloud.

Current research interest grows towards in-cloud media processing instead of having the contemporary flavour of using separate storage and processing units. To do so, Active Media Store (AMS) [125] extends OpenStack Swift to enable computations such as transcoding, automated metadata extraction, quality checks, etc. However, it does not focus on how to deploy secured media cloud along with media processing methodology. Besides, another study [126] designs a software platform based on Google cloud that supports various video analysis tasks in addition to transcoding. Additionally, a recent study [53] shows deployment of a secured private cloud for image resizing and storing using a middleware named iBuck on OpenStack Swift. Focus of this study only covers images and not extendable for other media files such as videos.

4.3.2 Literature on Video Surveillance System

One of cloud computing's five essential characteristics is on-demand self-service which ensures cost efficiency as the user is only paying for the services he or she is using [127]. Furthermore, in study [128], the authors proposed an on-demand cost-efficient method for storing video streams in the hierarchical storage of the cloud which enables the method to decide the video streams that should be pre-transcoded as it requires a large storage space. Which can be costly and this method can minimize the cost up to 40% and this method talks about video streams that can be divided into many sequences which are created by Group Of Pictures (GOP). In addition to mobile-based video streaming, [129] proposed an efficient framework for smooth and high quality video transmitting based on DASH protocol for personal cloud such as Dropbox, OneDrive. Study [130] proposed a cloud-based Video on Demand(VOD) system that can ensure concurrency and scalability. This is much needed for this sort of environment that can be used as computers or smartphones, to continue providing services around the globe.

Cloud-based Video Storage System (CVSS) is one of the most extensively used services of cloud system which brings the concern of security and privacy policies of cloud-based systems. Study [131] provided a method that is able to provide security for CVSS. With the increasing use of video data, most organizations prefer CCTV footage or video data monitoring for security purposes and it rises the necessity of constant monitoring which can be quite challenging. In study [114], the authors proposed a three-layered cloud-based video monitoring and analysis system that address these problems such as processing time, human monitoring, GPU space, and so on and seeks to find a solution. Studies [11,53] proposed an architecture named SPMS through expanding cloud file-sharing capabilities. Real-time

Cloud-based people counter system is proposed for security, tracking, and marketing purposes using Raspberry Pi embedded system [132]. Study [133] proposed a method to find a certain person through a video query system using cheap object detection and priority ranking to analyze relevant video. Here, they proposed an in-storage media processing system along with performing encryption-decryption on user demand. This Chapter presents a new methodology that can utilize cloud media file-sharing capabilities beyond just storing files.

CCTV footages can be troublesome to deal with for face recognition as it can be affected by the footage quality or rain or expression, clothing, light, and so on. Studies [23, 113] propose an access control method based on video surveillance. It also includes a face recognition system based on CCTV machine learning using Radio Frequency Identification (RFID). Study [22] proposed a new method named deduplication which uses hashing to verify if the data is deduplicated.

But biometric authentication and face recognition can be quite challenging in order to secure individual privacy and identification [134]. Besides, for streaming data, study [135] proposed a method of RealEdgeStream (RES) an edge enhanced stream analytic system with filtration and identification phases.

As most of the CCTV environment deals with large video files, storing them in separate storage efficiently will be a challenging task. In study [136], we can learn how to process a video on a cloud platform including uploading and downloading footage and converting in different resolutions, segmenting, and retrieving the whole file. Study [11] created a Secure Processing-Aware Media Storage, then evaluated different types of video quality, bitrate, resolution, duration, and size by comparing in two different ways (subjective and objective evaluation).

Moreover, to the best of our knowledge, these studies are yet to focus the approach of using private clouds to process media files along with enhancing security, efficient storing and long-term archiving of video surveillance data. Moreover, the approach of using two different servers for better availability and ensuring fault tolerance using OpenStack Swift is yet to be focused in the literature. In this Chapter, we attempt to focus this approach for secure in-cloud media processing based on Swift.

4.4 Our Proposed Methodology

We propose to use a private cloud in place of using public clouds, where media processing is done in the cloud-storage. Figure 4.2a presents the architecture of conventional storage services and Figure 4.2b presents our proposed architecture for storing and retrieval of media files from SPMS server. As the figures depict, the basic difference in performing operations in our proposed model is to enable off-line preprocessing of media files. Thus, we name our proxy server (as shown in Figure 4.1 as proxy process) as proxy-media server. Here, we propose to add three different middleware services namely PhotoPool, MediaBucket, and SecureCloud on OpenStack Swift proxy server for image processing, video transcoding, and performing encryption-decryption tasks.

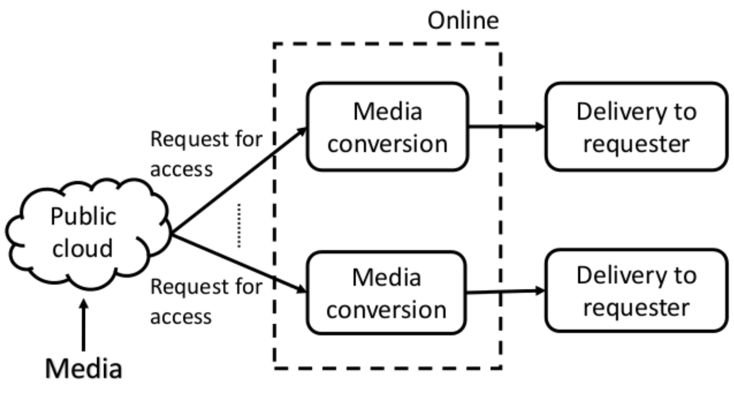
Figure 4.4 presents the methodology of our proposed system. Where we first upload a file to the system. If the file is a regular media file we forward it to the SPMS server and do the media conversions. If the file is a surveillance data then we push the file to process as a surveillance file. We first split the files then do conversions using FFmpeg, then store the metadata in the database. Finally we store all the files in the storage and archive servers as archival is necessary for surveillance data. Our proposed operational methodology over this architecture comprises several key steps, which we present in the following subsections.

4.4.1 Image Processing

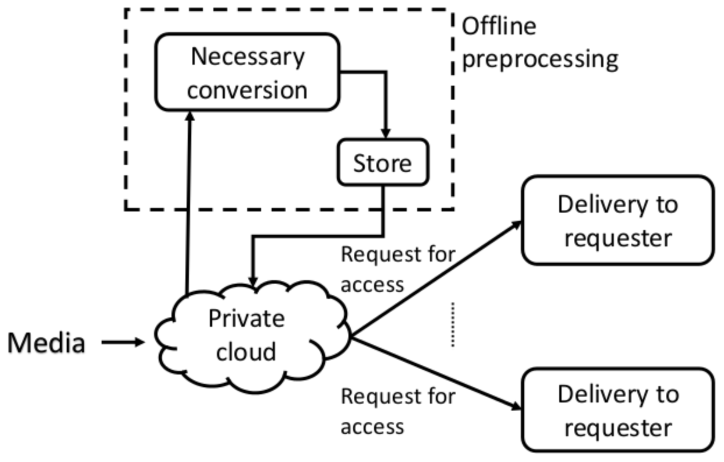
We add a new middleware named PhotoPool for only image processing and resizing (based on PIL algorithm). Security related tasks are performed by another middleware for both image and video type files. Figure 1 and 2 describe algorithms for uploading media files and PhotoPool middleware. Note that in Figure 2, we use an algorithm from [53], which was our prior study focusing only on image files.

4.4.2 Video Transcoding

In conventional model of video processing server, all compression and processing are done on different video servers and then all video files are sent to the proxy for uploading. This can cause substantial loss of time and bandwidth. Figure 4.3a presents the conventional model of processing video files outside the cloud storage. In contrast, we propose to do the processing related tasks in the proxy-media server through adding a new middleware MediaBucket. Here, users can transcode videos according to



(a) Conventional model

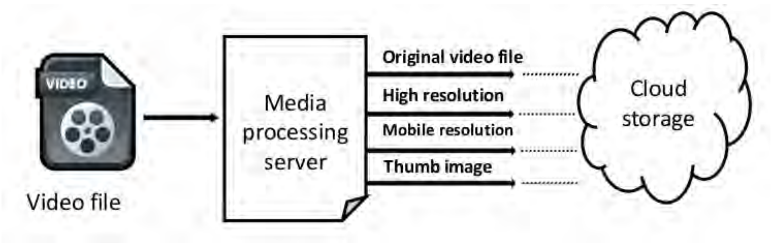


(b) Proposed model

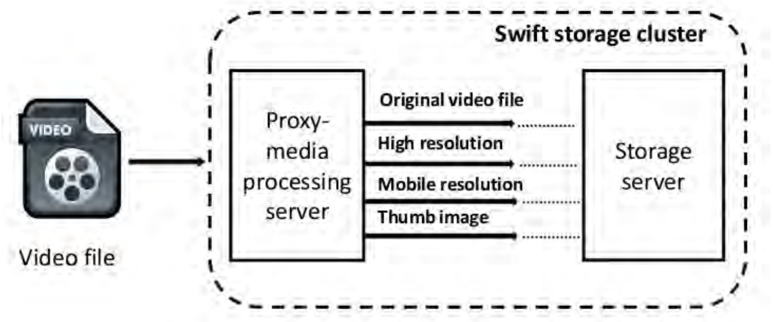
Figure 4.2: Conventional and proposed models for storing and retrieving media files from the cloud

several resolutions. We consider transcoding videos in two resolution: high resolution (720 to aspect ratio) and mobile resolution (480 to aspect ratio), which can be extended for other cases. Figure 4.3b presents our proposed model for uploading video files along with several transcoded versions. In our model we adopt FFmpeg tool for video transcoding, which can transcode good quality video with optimized resolution and size [137]. Figure 3 presents the algorithm of MediaBucket middleware. Here, we also propose two scenarios for uploading video files based on file size and user demand. Though, we transcode videos in several resolutions using FFmpeg tool, for larger file there may have chances to occur client timeout. Hence, users can choose an appropriate algorithm from the following options based on their requirements.

Response after all uploading: In this scenario, users get success response only if all versions of a



(a) Conventional model of processing videos outside the cloud storage



(b) Proposed model of processing videos inside the cloud storage

Figure 4.3: Conventional and proposed models of processing media files

media file are successfully uploaded.

Quick response with background processing: In this scenario, users get a response quickly after uploading the original video file and other processing tasks are done on background. After uploading all other transcoded video files, server notifies the users as per the query head request.

Figure 4.5 and 4.6 present flow diagrams of the proposed two scenarios as mentioned above.

4.4.3 Media Security

We use PFCC (P-Fibonacci transform of Discrete Cosine Coefficients) algorithm for encryption-decryption of image files [138]. Besides, encryption-decryption of video files are done while on transit from proxy-media server to storage server, as data is encrypted with SSL/TLS certificate here. We propose to use pycrypto module [139] for the perpose of video encryption-decryption. Thus, users can use pycrypto any cipher, eg. AES, ARC2, ARC4, Blowfish, CAST, DES, DES3, PKCS1_OAEP, PKCS1_v1_5 and XOR, in different modes such as CBC, CFB, CTR, ECB, OFB, etc., based on their needs. We implement all tasks related to encryption-decryption in SecureCloud middleware. Figure 4 describes the algorithm of SecureCloud middleware. Here, we consider only AES cipher for the

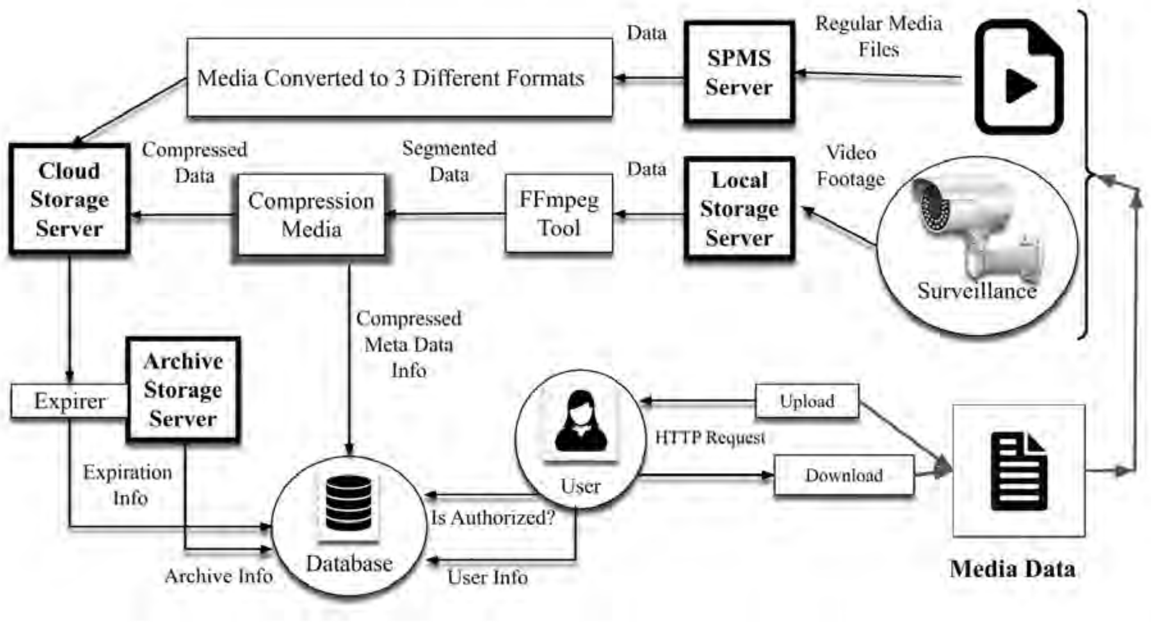


Figure 4.4: Our proposed model comprises SPMS server for media files storing and retrieval, and archival server for CCTV surveillance system

purpose of video encryption-decryption even though other alternatives could have been chosen.

This part focuses on how we can store the video files into our servers and archiving them as part of the process. For this we use two servers: the first server will be a storage server which will store the raw video files and after a certain period, one month, in this case, the video file will be automatically deleted. The second server will be an archive server where the files will be stored indefinitely in compressed form. The reason for selecting two different servers is consumers may require some files immediately after a certain incident had occurred. There is no compression in the storage server which ensures better performance as the videos are uploaded simultaneously in two servers and the archive servers already requires compression. Any kind of compression in the storage server may result in hardware throttling of the client machine as video compression requires a lot of processing power. They can access the storage server with ease and retrieve uncompressed and untampered data. But they might wish to access past videos for varied reasons, then they can retrieve their data from the archive server.

Algorithm 1 Algorithm for uploading media file

```
1: procedure UPLOADMEDIA($app, media, path, env$)  ▷ Uploading media to destination path using application and environment
2:     $MAX\_FILE\_SIZE \leftarrow 5Gb$
3:     if ($media > MAX\_FILE\_SIZE$) then
4:         raise uploadError
5:     end if
6:     $new\_env \leftarrow createNewEnv(env)$                ▷ Function of Swift
7:     $req \leftarrow createObjectReq(path, new\_env)$        ▷ Function of Swift
8:     $response \leftarrow req.getResponse(app)$             ▷ Function of Swift
9:     if ($response \neq SUCCESS$) then
10:        raise uploadError
11:    end if
12: end procedure
```

4.4.4 Surveillance System

To make a surveillance system, we use a good IP/CCTV camera because resolution depends on the number of pixels in the CCD (Charge-coupled device) chip. In Table 4.1 we present the resolution used in CCTV cameras [16]. In other words, the resolution is directly proportional to the number of pixels in the CCD chip [140]. Additionally, CCTV is much lower in terms of resolution than IP cameras. The resolution of analog cameras results in very small field of view when compared to IP cameras from an old evaluation. Our proposed surveillance system comprises several key steps, which we present in the following subsections.

Table 4.1: Cameras resolutions used in CCTV surveillance system [16]

Designation	HxV
CIF	352x240
2CIF	704x240
4CIF	704x480
D1	720x480
720p HDTV	1280x720
1.3MP	1280x1028
2MP	1600x1200/1920x1080

4.4.4.1 Local Storage

We use Network Video Recorder (NVR) or Digital Video Recorder (DVR) for storing the footage which depends on the user's demand. NVR is the more powerful storage system for storing data compared to DVR and NVR is recommended for long-distance areas. Before going to the cloud

Algorithm 2 Algorithm for the PhotoPool middleware

```
 1: procedure PHOTOPOOL($app, env$)
 2:     $origImage \leftarrow OriginalImage$
 3:     $resize\_dimensions \leftarrow UserGivenDimensions$        ▷ Example = [150, 300, 600]
 4:     $path \leftarrow UploadedPathFromEnvironment$
 5:     if $requestMethod \neq PUT$ then
 6:         return app
 7:     end if
 8:     if $original\ image\ not\ in\ right\ format$ then
 9:         $im.save(origImage, JPEG)$        ▷ PIL library function
10:     end if
11:     $UploadMedia(app, origImage, path, env)$
12:     $im.save(origImage, JPEG, prograssive = True)$
13:     $UploadMedia(app, origImage, path, env)$
14:     for $d$ in $resize\_dimensions$ do
15:         $i \leftarrow ResizeImage(d, origImage)$
16:         $UploadMedia(app, i, path, env)$
17:         $im.save(i, JPEG, prograssive = True)$
18:         $UploadMedia(app, i, path, env)$
19:     end for
20:     return app
21: end procedure
```

storage part, we have to consider few more things for efficiency as well as a cost-effective system.

Table 4.2: Calculation of DVR and NVR

Storage Type	DVR	NVR
Camera Stream	H.264	H.264
Camera Resolution	2 MP(1920x1080)	2 MP(1920x1080)
Video Quality	Medium	Medium
Average Frame Size	13.714 KB	21.2KB
Frame Rate	15	15
Hours/Day	24	24
Bandwidth Required	2.4 Mbps/Cam	2.6 Mbps/Cam
Desired days of Storage	1	1
Estimated Storage	16-37GB(Approx.)	18-28GB(Approx.)

HDD, flash drive, SD card, SSD can be used as storage device for NVR. Since we focus on large footage, we choose HDD for storing the footage as in Table 4.2 shows the expected storage for DVR and NVR. We calculate the values on Seagate and SuperCircuits website.

Algorithm 3 Algorithm for the MediaBucket middleware

```
1: procedure MediaBucket($app, env$)
2:     $origVideo \leftarrow Originalvideo$
3:     $vInfo \leftarrow OriginalVideoInformation$    ▷ Get all video related information [137]
4:     $resize\_resolutions \leftarrow UserGivenResolutions$    ▷ Example = [720, 480]
5:     $path \leftarrow UploadedPathFromEnvironment$
6:     if $requestMethod \neq PUT$ then
7:         return app
8:     end if
9:     $UploadMedia(app, origVideo, path, env)$
10:    for $r$ in resolutions do
11:        $v \leftarrow TranscodeVideo(r, origVideo, vInfo)$    ▷ Transcode video from original video using FFmpeg command based on algorithms given in Fig. 4.5, 4.6
12:        $UploadMedia(app, v, path, env)$
13:    end for
14:    $tImage \leftarrow ThumbTranscode(origVideo, vInfo)$    ▷ Create thumb image from video using FFmpeg [137]
15:    $UploadMedia(app, tImage, path, env)$
16:    return app
17: end procedure
```

4.4.4.2 FFmpeg Media Converter

We chose to segment our files before compressing and uploading them because in case of network failure the file will get damaged. This is also more convenient for the users because they request a certain footage, they have the ability to download a smaller video file from the specific time they need instead of the whole 24-hour footage. We use the following command to convert the file into 30 minutes segments using ffmpeg presets.

ffmpeg -i input.mp4 -c copy -map 0 -segment_time 1800 -f segment -c:v libx264 -preset veryslow output%03d.mp4

FFmpeg is built with a number of self-contained libraries which provide discreet functionality that can be included in other applications. These features include codec encoding and decoding, compression, image scaling, resampling, and format conversion.

4.4.4.3 User Interface

Our proposed architecture is designed for simplicity and high effectiveness. Hence, we design a user-friendly interface that is both intuitive and provide fast results. Most of the task, which starts from

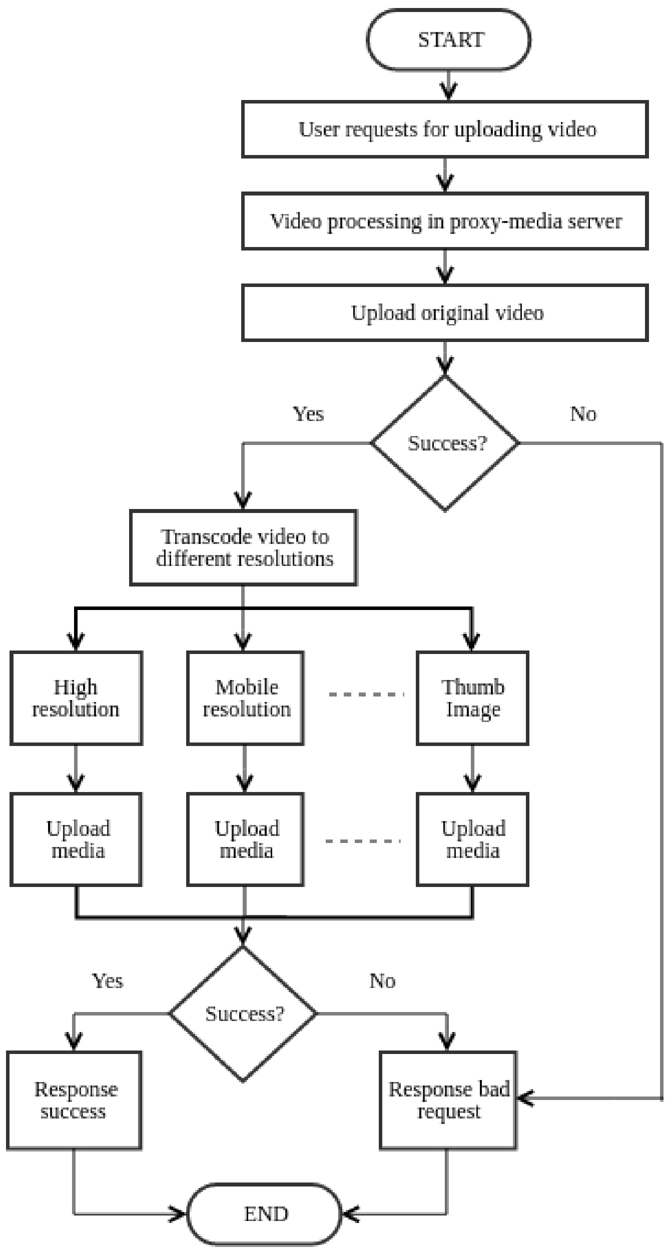


Figure 4.5: Flow diagram of video uploading and transcoding for 'Response after all uploading' scenario

getting the video data from CCTV to splitting to processing and compression; most of these are handled automatically on the cloud by the system. Very few times when clients may try to connect to the storage manually. To this extent, we design an user interface (Figure 4.7) where clients can get

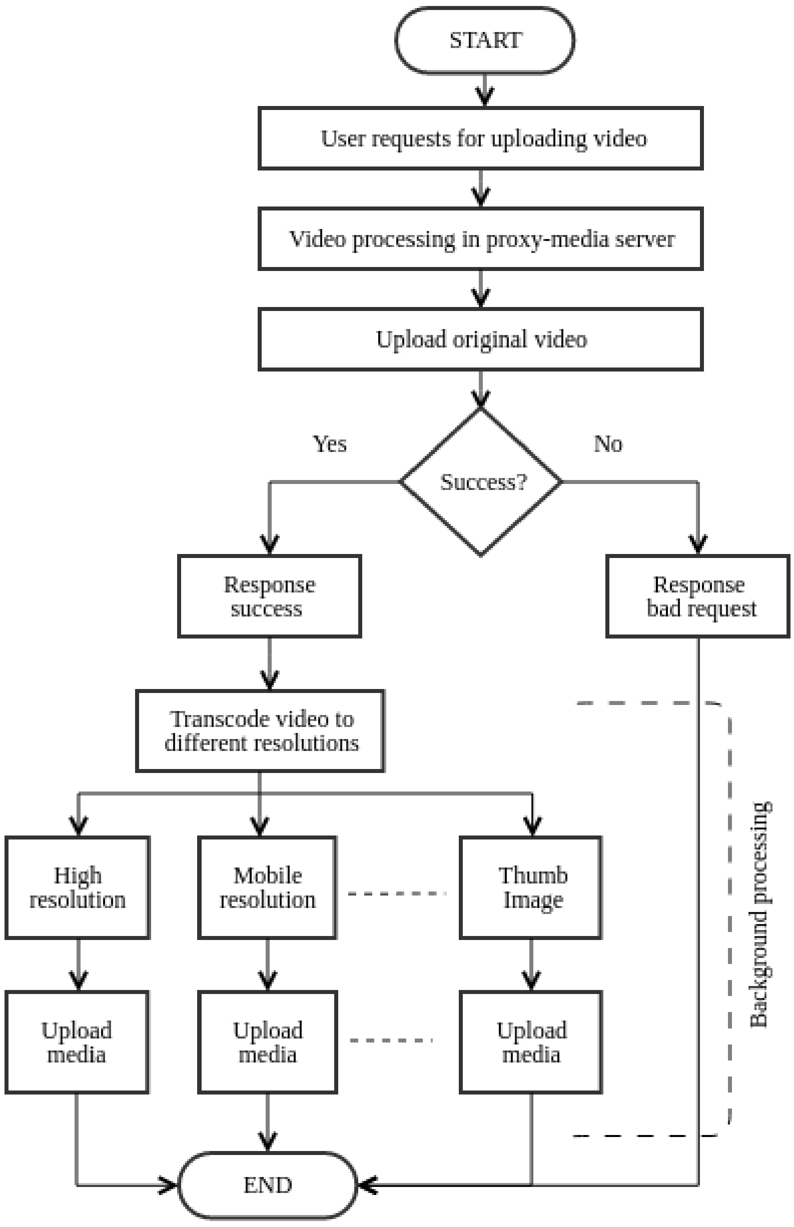


Figure 4.6: Flow diagram of video uploading and transcoding for proposed ‘Quick response with background processing’ scenario

the job done without any technical understanding of the entire video storage system.

Algorithm 4 Algorithm for the SecureCloud middleware

```
procedure SecureCloud($app, media$)
    if $media = image$ then
        if $requestMethod = PUT$ then
            $EncryptPFCC(media)$                                  ▷
        end if
        if $requestMethod = GET$ then
            $DecryptPFCC(media)$                                  ▷
        end if
    end if
    if $media = video$ then
        if $requestMethod = PUT$ then
            $EncryptAES(media)$        ▷ Encryption algorithm is given in [139]
        end if
        if $requestMethod = GET$ then
            $DecryptAES(media)$        ▷ Decryption algorithm is given in [139]
        end if
    end if
    return app
end procedure
```

4.4.4.4 Object Expiration

Object expiration is Swift's built-in feature, which is designed to expire an object. It automatically deletes objects after a given time that the user can set. We adopt the expiration feature in our system to ensure efficient storage. In our architecture, after we upload an object to our server, we pass an expiration header $X - Delete - At$ or $X - Delete - After$ header with a POST request. After the time we send in the header passes, it automatically deletes that particular object. Besides, once the object is deleted, Swift will no longer serve the object and it will be deleted from the cluster shortly thereafter.

The $X - Delete - At$ header takes a Unix Epoch timestamp, in integer form. The $X - Delete - After$ header takes a positive integer number of seconds. The proxy server that receives the request will convert this header into an $X - Delete - At$ header using the request timestamp plus the value given. Figure 4.8 presents the expiration feature of the proposed architecture. This command is used to use the expirer middleware for the archive storage:

swift -A http://127.0.0.1:8080/auth/v1.0/ -U test:tester -K testing post archive_storage sample_video.mp4 -H "X-Delete-After:2592000".

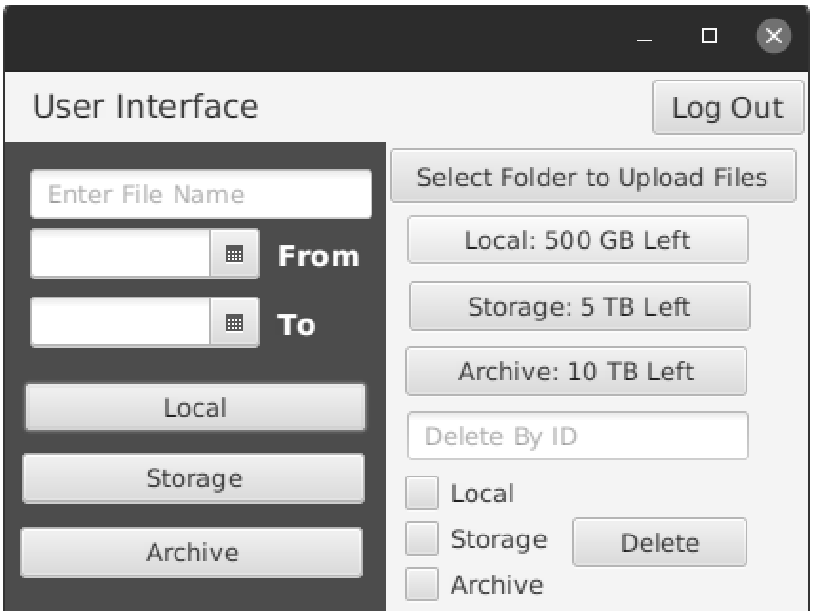


Figure 4.7: Designed user interface for uploading and downloading CCTV surveillance data

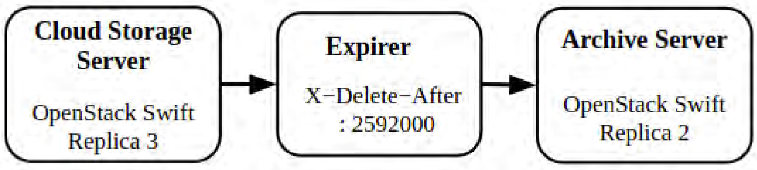


Figure 4.8: Expiration of an object using the $X - Delete - After$ header when uploading a segmented CCTV footage

4.4.5 Storage Server

Figure 4.9 shows the three different storage servers in our architecture - local storage, storage server, and archive server. The storage server functions as follows: The video will be uploaded unmodified and unchanged with the expiration header set to "$X - Delete - After$: 2592000". Which means it will be deleted automatically after a month. We have set the replication count to three when we constructed our ring so if one server fails, users will still have two backups of their files. This will ensure the safety of their files.

4.4.6 Archive Server

In the archive server, we first split the files using FFmpeg, compress them, and then upload it with no expiration headers. We set the replication count to two here since the data here will increase over

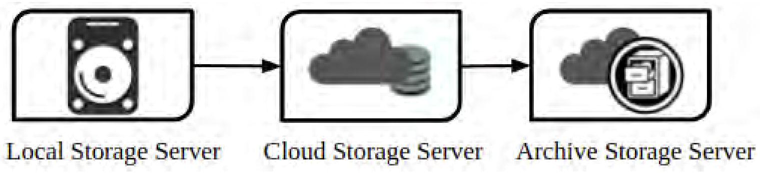


Figure 4.9: Three types of storage servers for ensuring high availability, less storage, and long-lasting archive system. Local storage server has the direct connection with CCTV camera. Cloud storage and archival storage servers are located in data centers.

time and keeping more replicas will become more and more expensive as time goes on. Replication count two also ensures better data availability than one as in case of one server or storage node fails, user still have a backup for all the data [141]. Since it is an archive server, a replication count of 3 is going to be too expensive as the number of files increases.

4.5 Experimental Evaluation and Comparative Analysis

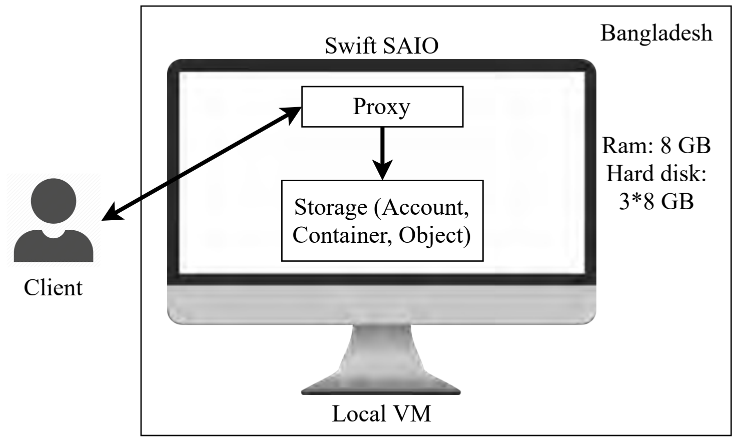
We evaluate performance of our proposed methodology through a real implementation. Besides, we compare the performance against that obtained from conventional methodology, as shown in Figure 4.3a. Before presenting the evaluation results, we first briefly elaborate our experimental settings.

4.5.1 Experimental Settings

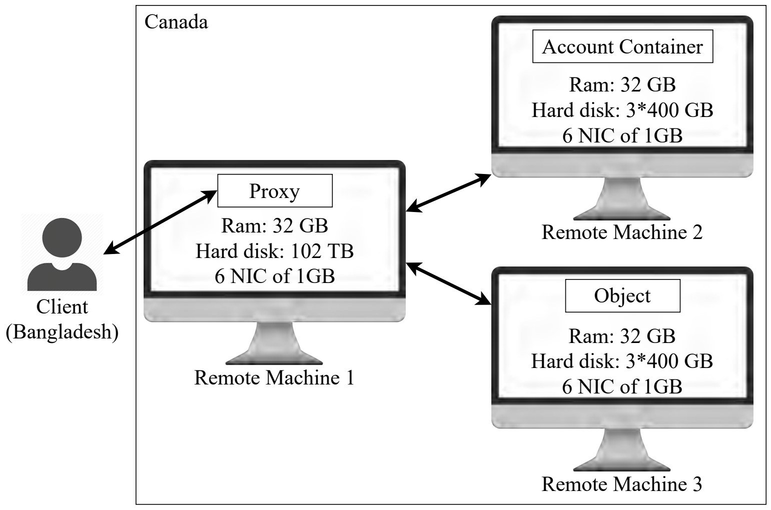
We use two different server setup for evaluating our SPMSA system. At first, we present the server configurations, media files information of SPMS system testbed. After, we discuss the server settings of CCTV surveillance system.

4.5.1.1 Testbed for SPMS Server

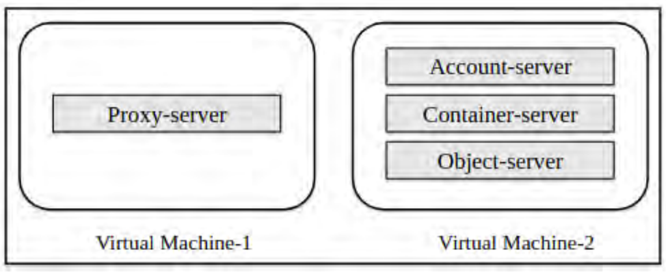
Our experimentation covers transcoding of different video files in different resolution minimizing size, time, and bit rate along with maximizing the video output quality to the end user. Here, we choose client machine located in Bangladesh having configuration of Intel(R) Core(TM) i3-4150, CPU 3.50GHz, and memory 16 Gb for transcoding and uploading videos. We chose several video files having different sizes, however, with same video resolution. We transcode the chosen video files at different resolutions varying FFmpeg command options such as profile, level, preset, constant rate factor, frame rate, X264opts, etc. Table 4.3 shows the informations of files used for quality measurement.



(a) Local server setup in virtual machine



(b) Remote server setup



(c) Multi-node Swift setup

Figure 4.10: Server setup for our proposed framework for testing SPMS server [12] (in 4.10a and 4.10b). Figure 4.10c presents the multi-node Swift setup used for CCTV surveillance system.

We evaluate the quality of videos in two different ways:

- **Subjective evaluation:** Video is used to be watched by people. Therefore, this method is

Table 4.3: Information of files used for video quality testing

Quality test file	Size (Mb)	Bit rate (kbps)	Resolution	Duration (min)
File 1	36	1061	1280 × 720	4.07
File 2	54	2291	1280 × 720	3.06
File 3	66	1873	1280 × 720	3.34

Table 4.4: Information of files used for uploding in several storage services

Upload test file	Size (Mb)	Bit rate (kbps)	Resolution	Duration (min)
File 1	18	583	320 × 240	4.07
File 2	36	1061	1280 × 720	3.06
File 3	98	1836	1280 × 720	3.34

more suitable to evaluate quality of transcoded videos. We invited 20 observers to observe videos and calculate MOS based on their evaluations [11].

- **Objective evaluation:** We compressed File 2 (as shown in Table 4.3) to 480 × 270 resolution using different preset values for obtaining corresponding time, sizes and bit rates. We utilize a video quality measurement tool [15] to calculate QoE metrics, i.e., MSE, PSNR, SSIM, and VQM.

Furthermore, we evaluate the time needed for storing video files in our proposed SPMS server. Table 4.4 shows information of the video files used for uploading time performance measurement. We perform the uploading through conducting experiments in two different setups. One setup is in local machine in Bangladesh, where servers are setup in a local virtual machine. Another setup is in a remote data center in Canada. For each setup, we install three different servers - one for proxy, one for account-container, and the last one for object server. The memory and disk configurations of testing servers are as follows:

- **Local SPMS Server:** Proxy, account-container, and object servers are installed in a local virtual machine having 8 Gb memory and 3 disks each of 8 Gb (in Figure 4.10a). The machine is network connected through bridge connection.

- **Remote SPMS Server:** The remote server consist of one proxy having 32 Gb memory and 1.2 Tb disk, one account-container having 32 Gb memory and 3 disks each of 400 Gb, and one object having 32 Gb memory and 3 disks each of 400 Gb. Each server has six 1 Gb network interface cards (in Figure 4.10b).

Furthermore, we setup dropbox and google drive in our local client machine in Bangladesh. The uploading and average network speed were around 1.95 Mbps and 1.5 Mbps respectively when we conduct our experiments.

4.5.1.2 Testbed for Surveillance System

Our experimentation includes the comparison of videos segmented and compressed with FFmpeg along with the bit-rate of various presets of FFmpeg tools. Here, we conduct our video conversion and segmentation experiment in a client machine located in Dhaka, Bangladesh having a specification of Processor Intel Core i7 9850H, CPU 2.6-4.6 GHz, Memory 16 GB, and GPU nvidia 1050. Besides, we compare different presets of FFmpeg tools.

Furthermore, we set up our own storage server and archive server in the remote machine located using Google Cloud Platform (GCP). We store these experimental videos on our server. We perform the uploading through conducting experiments in a setup that is in a multi-node setup, where we install a multi-node Swift with two different servers - one for proxy server another one for account, container, and object server [4]. We calculate the average metrics of these setup while evaluating the proposed architecture. The memory and disk configurations of testing servers are as follows:

- **Multi-Node Swift Setup:** We have a total of four virtual machines in this setup. The storage and archive server each has two virtual machines where the proxy server of Swift is running in one server and the other three account, container and object servers are running on another virtual machine. The virtual machines have 4GB of memory and 100GB of storage each. The OS is Ubuntu 18.04 server for all the virtual machines. In Figure 4.10c we can see the setup for one of the archive and storage server which are identical.

In our storage nodes, we use Swift replication count of 3 and in archive nodes we have used a replication count of 2.

4.5.1.2.1 Experimental Data

In order to evaluate our proposed architecture we use a benchmark video surveillance data set from the VIRAT [142]. We use 125 videos ranging in size from 3.8MB to 1.4GB. Figure 4.11 presents the

Table 4.5: Comparison of size, bit rate, and time needed for different preset option of FFmpeg command

Preset option	Notation	Size (Mb)	Bit rate (kbps)	Time (s)
Medium	P_m	19.0	736	35.3
Fast	P_f	19.1	738	33.9
Faster	P_{fr}	18.8	725	26.4
Faster, Subme=6	P_{frs6}	19.2	743	31.0
Veryfast	P_v	18.7	724	23.8
Veryfast, Subme=6	P_{vs6}	18.7	724	31.1
Superfast	P_s	22.7	902	24.6
Superfast, Subme=6	P_{ss6}	19.0	734	25.0
Ultrafast	P_u	30.1	1234	21.1
Ultrafast, Subme=6	P_{us6}	21.4	844	24.4

video files which are used to test the efficacy of the proposed architecture. The different sizes of the video files are used to evaluate the proposed systems performance for different scenarios.

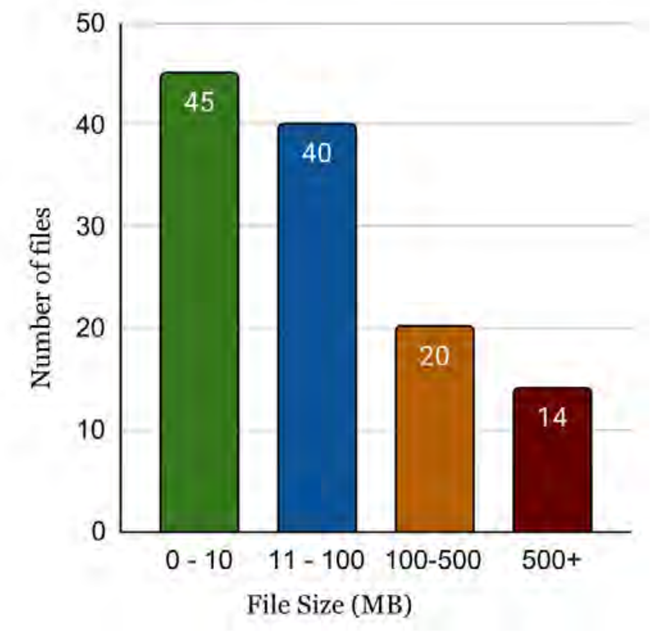


Figure 4.11: Video data used in experiment

Table 4.6: Size, bit rate, and time comparison of different files for two FFmpeg preset options

File	Resolution	Veryfast			Superfast, Subme=6		
		Size (Mb)	Bit rate (kbps)	Time (s)	Size (Mb)	Bit rate (kbps)	Time (s)
File 1	480×270	11.3	267	23.1	13.3	332	26.3
	560×316	13.4	337	24.4	15.9	420	29.1
File 2	480×270	18.7	724	20.2	19.0	734	20.9
	560×316	23.4	934	22.8	23.8	952	23.6
File 3	480×270	17.9	430	30.6	20.0	493	32.6
	560×316	22.1	557	34.1	24.9	645	36.0

4.5.2 Experimental Results

Table 4.5 shows the size, bit rate and time needed for compressing File 2 (in Table 4.3) into 480×270 resolution using 3 different preset values and FFmpeg options.

Here we set values of profile, level, constant rate factor, and frame rate to baseline, 3.0, 25 and 24 respectively. From observations of participants of our experiment, we calculate MOS for each transcoded videos.

Table 4.5 presents that both preset option veryfast and superfast with subme value 6, provide lower values of size, bit rate and time after compression. Besides, MOS results for this two preset options are almost same. Accordingly, we transcoded three different files given in Table 4.3 into two resolutions 480×270 and 560×316 using these two chosen preset options for finding the better preset. In every case, compression using veryfast preset option provides lower size, bit rate and time needed compared to that obtained using superfast with subme value 6. Table 4.6 shows the size, bit rate and time needed for compression of different files. We run the commands for 20 times and present their averages as our experimental results.

For obtaining the QoE values through objective test, first, we transcode the File 2 (in Table 4.3) into 480×270 resolution without changing any other FFmpeg options. Size and bit rate of this new file are 20.8 Mb and 798 kbps. We find its transcoded time to be 43.5 s. We use this file for QoE comparison. Table 4.7 shows the values of MSE, PSNR, SSIM, and VQM comparing each transcoded video files with the new compressed file. This table shows that the video file transcoded using veryfast preset option exhibits higher quality than superfast preset option with subme value 6.

Table 4.7: QoE measurement of MSE, PSNR, SSIM, and VQM for different preset options of FFmpeg command

Preset option	MSE	PSNR	SSIM	VQM
Medium	12.13	37.29	0.97	0.94
Fast	12.17	37.28	0.97	0.94
Faster	11.98	37.35	0.97	0.93
Faster, Subme=6	12.16	37.28	0.97	0.94
Veryfast	14.09	36.64	0.96	1.02
Veryfast, Subme=6	12.87	37.03	0.96	0.98
Superfast	14.50	36.52	0.96	1.02
Superfast, Subme=6	15.52	36.22	0.96	1.04
Ultrafast	17.42	35.72	0.94	1.14
Ultrafast, Subme=6	16.24	36.03	0.94	1.05

Table 4.8: Comparison of required time and CPU usage for two scenarios in the local server

Upload test file	1st scenario		2nd scenario	
	Uploading time (s)	CPU usage (avg)	Uploading time (s)	CPU usage (avg)
File 1	23.8	98.2	23.6	98.2
File 2	91.7	98.2	91.4	98.2
File 3	185.6	99.3	185.6	99.3

Table 4.9: Comparison of required time and CPU usage for two scenarios in the remote server

Upload test file	1st scenario		2nd scenario	
	Uploading time (s)	CPU usage (avg)	Uploading time (s)	CPU usage (avg)
File 1	92.6	16.3	88.9	16.0
File 2	190.6	34.5	181.1	33.6
File 3	489.4	38.5	450.2	37.0

Afterwards, we test our proposed two scenarios (Figure 4.5 and Figure 4.6) for uploading video files. Table 4.8 and 4.9 show time required for uploading and average CPU usage during the uploading task for local and remote SPMS servers respectively.

As our local SPMS server configuration is not heavy weight enough and FFmpeg command needs higher processing, average CPU usage in the local server becomes almost 98%. Besides, timing difference in the local server is very small for both scenarios. However, in the remote server, average CPU usage is 15%-40% depends on file size, bit rate, etc. Here, uploading time decreases while using the 2nd scenario compared to the 1st scenario. The 2nd scenario is good for any size of video files.

Besides, for the 1st scenario, there is a substantial chance of getting timeout due to high processing time for large video files. If user wants to upload only small size videos, then the 1st scenario will be a good choice. Additionally, we compare uploading times for different video files both in Swift and

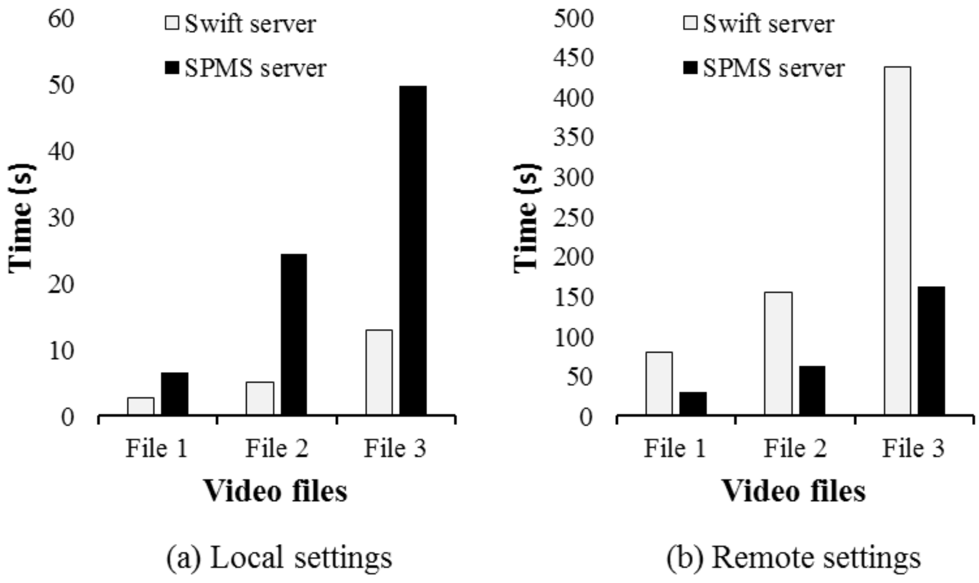


Figure 4.12: Time comparison for uploading videos in Swift and proposed SPMS server for both local and remote settings

in our proposed SPMS servers for local and remote settings. In Swift server, only original video files were uploaded. In proposed SPMS server, four different media files were uploaded for each video file. We take the average time of all the uploading for SPMS server. For the local server, uploading time is higher than the conventional Swift server and lower for remote server due to the server configuration and network latency. Figure 4.12 comprises the results.

We compare the uploading time of three different files given in Table 4.4 in our SPMS server along with conventional storage services such as Dropbox and Google Drive. Figure 4.13 shows that SPMS server always takes less time for video storing, than other conventional storage services. While storing, first, we transcode each file into different versions using the FFmpeg command in our local machine. Transcoding time for file 1, 2, and 3 are 19, 31 and 45 second respectively. Total time of uploading these transcoded videos in Dropbox and Google Drive is presented in this figure.

We use Linux based FFmpeg media converter tool to convert three videos into different resolutions with 3 different preset settings of FFmpeg and collect those data. The three presets are: default,

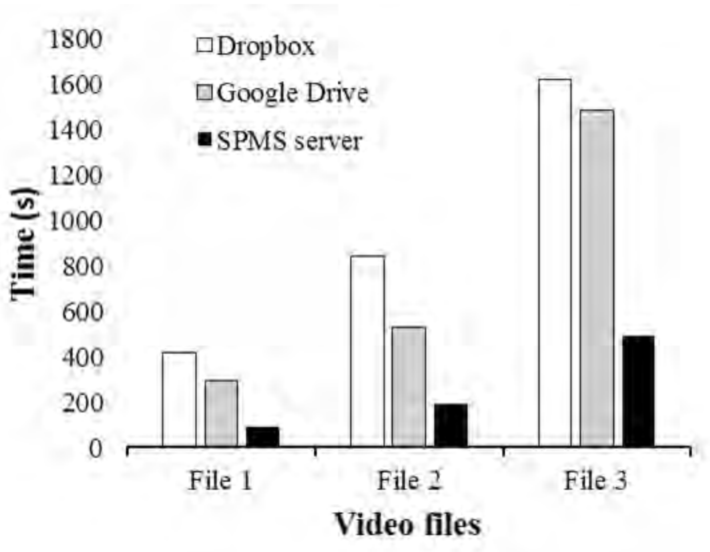


Figure 4.13: Time comparison of storing video files in conventional storage services and proposed SPMS server

veryfast and ultrafast. Figure 4.14 shows that veryfast preset method gives the best conversion results in terms of file size reduction.

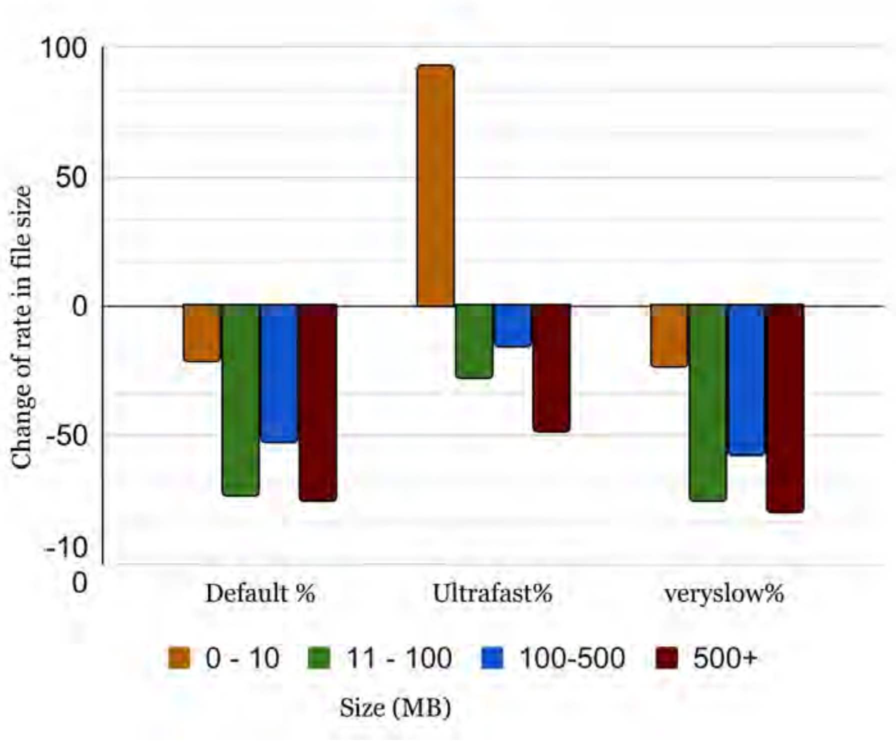


Figure 4.14: Rate of change of video size for different preset modes

If one has ample storage space for their servers he can choose ultrafast mode but if limited in storage it will be wiser to use veryfast preset mode of FFmpeg. Figure. 4.15 presents the time conversion for different sizes of videos from our data-set.

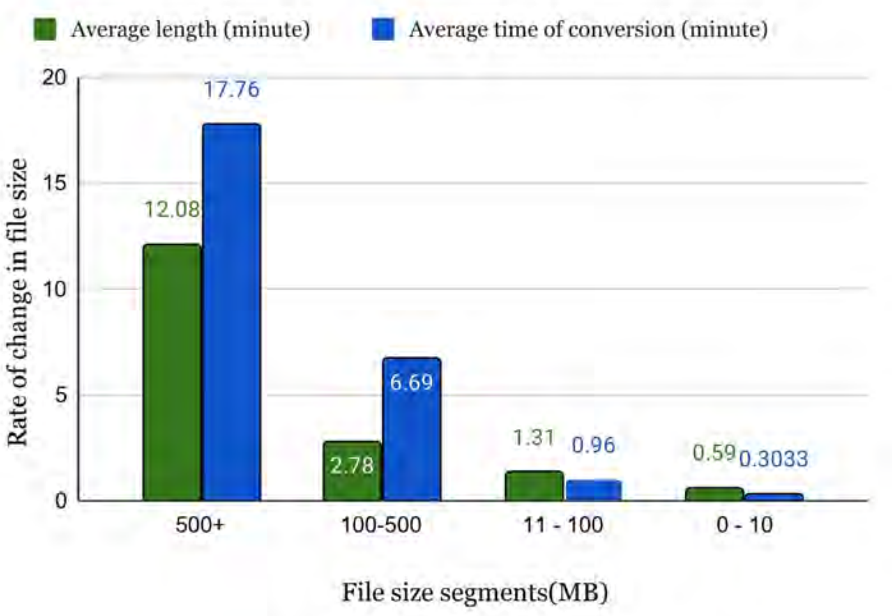


Figure 4.15: Comparison between time to convert and file length

From Figure 4.16 we can see the video quality before and after of the converted files.

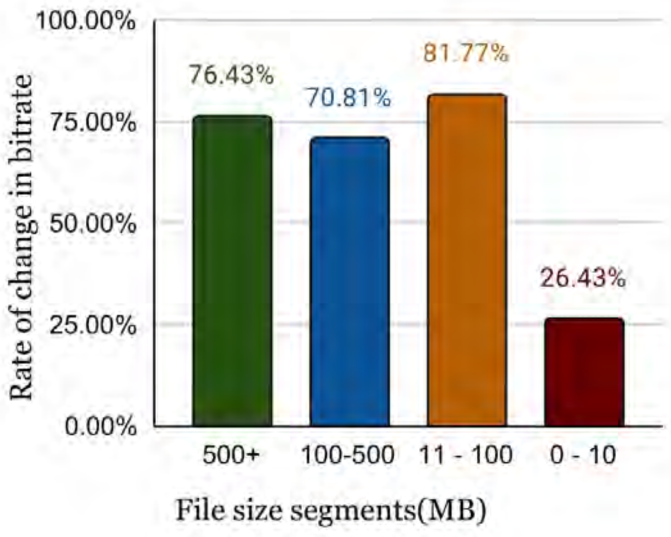


Figure 4.16: Comparison of video bit-rate before and after applying presets

Table 4.10: Time improvement of 2nd scenario over 1st scenario in the remote settings

Upload test file	Time improvement
File 1	4%
File 2	5%
File 3	8%

In Figure 4.17, we present the comparison between our proposed system, google drive, and iCloud.

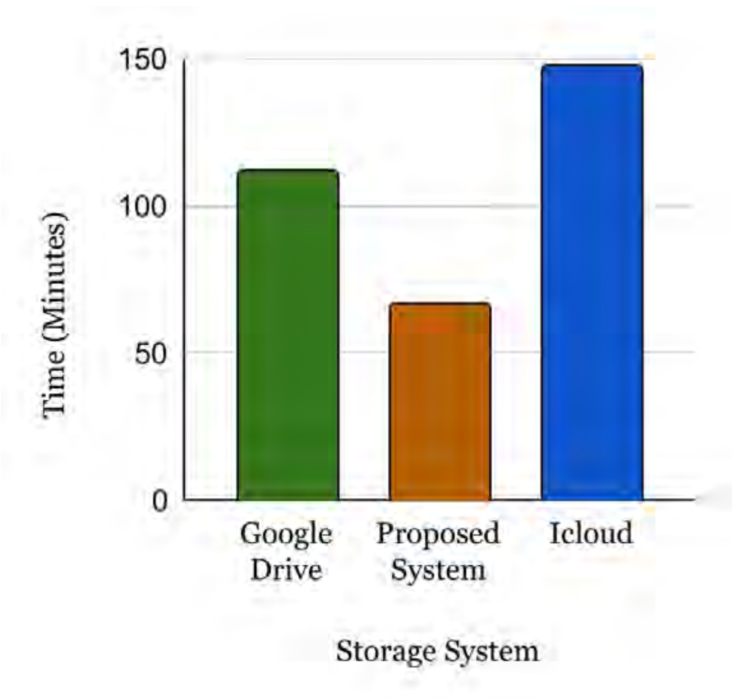


Figure 4.17: Comparison of content upload time between different storage systems

We conduct the tests five times with standard deviation of 3.92 for proposed system, 22.92 for google drive and 15.40 for icloud. We cap the bandwith to 30 megabits while uploading the files to different storage systems. If we compress the video data and store it in the archive server it takes less time. We use three replications on the storage server and two in the archive server. As a result storage server is taking more space than the archive server. Our proposed server shows significant improvement both in storing and archiving time.

4.5.3 Experimental Findings

Table 4.10 shows improvement in required time using the 2nd scenario compared to that using the 1st scenario in the remote server. Here, the improvement spans over 4%-8%. In the local server,

(a) Before - File Size less than 10 MB

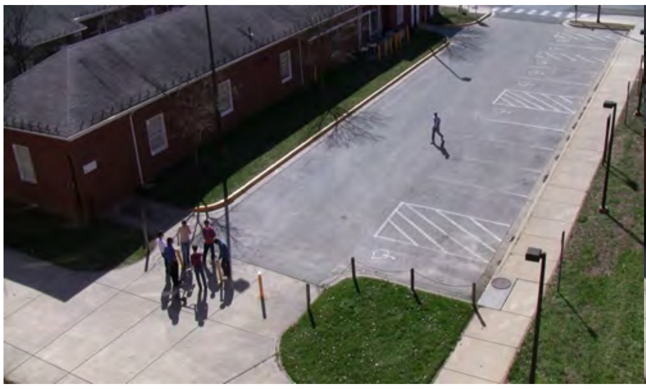
(b) After - File Size less than 10 MB

(c) Before - File Size less than 100 MB

(d) After - File Size less than 100 MB

(e) Before - File Size less than 500 MB

(f) After - File Size less than 500 MB


(g) Before - File Size more than 500 MB

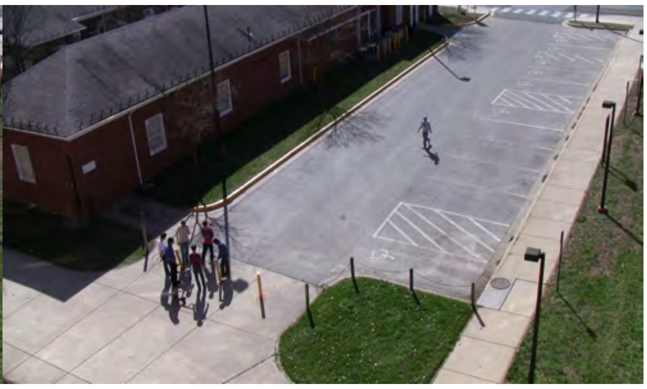

(h) After - File Size more than 500 MB

Figure 4.18: Comparison of video snapshots before and after applying FFmpeg presets

Table 4.11: Improvement in uploading time using proposed SPMS server compared to that using Swift server in the remote settings

Upload test file	Remote server
File 1	62%
File 2	60%
File 3	63%

Table 4.12: Improvement in uploading time using proposed SPMS server compared to that using conventional storage services

Upload test file	Dropbox	Google Drive
File 1	78%	67%
File 2	78%	64%
File 3	70%	67%

the improvement remains within 2%. Besides, Table 4.11 shows improvement in uploading time for remote servers with respect to conventional Swift server. Here, the improvement spans 60%-63% for the remote server. Besides, improvement of video file uploading time in remote server with using the proposed SPMS compared to that using conventional storage services such as Dropbox and Google Drive is given in Table 4.12. This table shows that users can save up to 78% and 67% time using SPMS server compared to Dropbox and Google Drive, respectively.

4.6 Discussion

Our proposed system solves some of the main issues with current implementations and achieves improved metrics [143] when compared to other popular implementations. Table 4.13 presents the theoretical comparison between On-site Physical Storage System [118], Google's cloud-based NEST Cam [121] and SPMS [11]. We present four challenges in the Introduction section. Our current implementations and proposed system provide the following solutions to address those challenges.

- Solution-1: We propose SPMS server for offline pre-processing rather than on demand processing. Though it costs more space but take less time and faster availability.
- Solution-2: Our proposed method promises high fault tolerance and availability, resulting in low equipment costs, as our proposed solution does not require any additional hardware upgrades and is compliant with existing implementations. Both physical storage [118] and NEST Cam [121] cost more than our proposed method and SMPS requires more storage as it keeps different versions of the files in different formats for faster access. Our proposed CCTV surveillance

Table 4.13: Comparison of various implemetation based on the metrics element (H=high, L=low, and A=average)

Methods	Physical Storage [118]	SPMS [11]	NEST [121]	Proposed
Performance	L	H	A	H
Response Time	H	A	A	A
Scalibility	L	H	H	H
Through Put	H	H	H	H
Reliability	L	H	A	H
Availability	H	H	L	H
Usability	H	H	A	H
Cost effectiveness	L	A	L	H

system doesn't require any new upfront cost as it can be operated on existing servers. If we take one video and store it on SPMS it will make three versions of the files in three different formats and store them. If we consider replication count three then we get nine copies in total for each file uploaded. On the other hand, our CCTV surveillance system only has a replication count of 3 for the storage server and a replication count of two for the achieve server. So we only get five copies for each file uploaded. This saves 45% of the storage need in the long run.

- Solution-3: Our proposed system does not need constant high bandwidth since we can use local storage as a buffer period. This is especially helpful for developing countries with poor internet connectivity. This ensures the high throughput of our system. SPMS shows improved results in terms of data accessibility as it stores multiple copies with different formats.

- Solution-4: Our proposed system stores all videos that ensure that no detail is lost since we do not use any event triggering approaches. We use FFmpeg to split files, making it easier to look for the desired files with a simple timestamp when needed. Unlike NEST and other cloud-based offerings which only store event-based video for a limited period of time which does not guarantee the usability of the data stored. Our system can segment large files and compress them which saves space and makes the data more usable when needed. This makes our system more efficient when compared to physical storage and cloud-based offerings. SPMS also provides efficient data retrieval methods.

Our proposed system is highly scalable without sacrificing lower cost, reliability, effectiveness, and efficiency.

4.6.1 Scalability of the Framework

Until now, we work with relatively small video files. In the real world scenario, however, the surveillance system may have thousands of cameras which would equate to a huge amount of video data. But because of the scalable and distributed architecture of OpenStack Swift's object storage, this will not be an issue. OpenStack Swift's cluster of servers can store petabytes of data. When a server begins to reach its storage capacity, we can add other servers in the ring file. This process can be repeated however many times we need, so it will always be scalable and expandable.

4.6.2 Reproducibility of the Framework

To deploy the system we need to have two object storage servers with OpenStack Swift installed and FFmpeg installed on the machine where the CCTV footage are being stored. Then the user just needs the interface we develop to upload and download the videos to their own private object storage system.

4.7 Conclusion

A key challenge for any large-scale computation today, whether in big data or in handling large-scale web services, is to guarantee efficient management as well as security of data. As media file sharing services are becoming capable of storing billions of objects distributed across server nodes, it becomes highly important to effectively manage as well as secure digital data of these services. Efficient storing and long-term archiving of video data is an important concern nowadays. However, these are not explored in the literature enough to date. Therefore, in this Chapter, we focus on this important topic pertaining to storage and archival of CCTV footage in the cloud. To do so, this Chapter presents a new methodology to expand cloud media file sharing capabilities from only storing media to perform sorting, resizing, and security tasks through moving and processing the data inside an object storage cloud. Our goal behind such a method is to extend the traditional role of object storage from being only a media repository to offering high availability and fast accessibility to secured data.

Here, we propose a new storing and long-term archiving methodology for CCTV footage leveraging the OpenStack Swift. Our proposed methodology cover all system-level aspects that need to be considered to deploy a real system. Accordingly, we deploy our proposed methodology in a real testbed. We perform rigorous experimentation over the deployed real testbed setup to evaluate efficacy of our proposed methodology. The evaluation results confirm that we can achieve substantial performance improvement using our proposed methodology.

There is plenty of room though to extent our study. Our future plans for such extensions include implementing customized encryption-decryption algorithms to take advantage of specified format of video files, in-cloud segmentation of video files to support adaptive streaming, partial transcoding algorithm of segmentation to minimize long-term cost, customized hierarchical PJPEG algorithm to reduce both store size at server side and loading time at client side, etc. Besides, we also plan to develop a new algorithm for transcoding the video with better quality in future. Furthermore, in future, we aim to integrate machine learning techniques into our system to detect unexpected behaviour in real time.

Chapter 5

A Novel Approach of Content-Based Searching in Object Storage System

5.1 Introduction

Tremendous amount of data is being produced every day which is stored and retrieved from various cloud servers. A recent estimation [144] shows people create about 328.77 million terabytes of data on a daily basis. In order to store these huge amounts of data, object storage is well-known to have an upper hand because of its flexibility and consistency. One of the distributed and consistent open-source cloud systems is OpenStack. It provides cloud object storage, allowing us to preserve and pull up large amounts of data using an API called OpenStack Swift. It is scalable and has been designed to be durable, and available for the whole data set. Swift is a well-suited storage system for unstructured data that can be increased immensely [12]. Swift object storage stores every single piece of data as an object, unlike the storage systems like file-based storage or block storage, which stores data as a file. This storage system is built to house massive amounts of data at a time because of its flexibility. The retrieval of relevant data has become a significant issue as the amount of data increases significantly [145]. Storing consumer and business data in either public clouds or private clouds has made it difficult to efficiently and effectively retrieve meaningful data [145].

As a result, cloud-based storage is being developed using object storage. Various well-known cloud service providers such as Amazon S3, OpenStack Swift, Caringo Swarm, and many others provide object storage. Although the problem of storing massive amounts of data is solved due to the complex architecture of object-based storage systems, retrieving or searching for a certain object/file has become a major challenge [146]. Object storage, for instance, OpenStack Swift uses the HEAD or GET method in order to get an object from the storage which is not very efficient when it comes to content-based searching. Moreover, the exact path of the object is also needed in order to get some data from this storage system, which is an inefficient task if there are massive amounts of data.

Even so, compared to block and file storage, object storage may produce higher delay and require more processing time, but it also has a number of advantages, including scalability, cost-effectiveness, robustness, and easier management. It offers great redundancy and data durability, making it particularly useful for managing massive amounts of unstructured data, another benefit of using Open Stack Swift as an Object Storage is, Swift object storage allows simultaneous access from several servers, therefore server binding is not a problem. A chaotic object storage system is present. It provides fault tolerance, scalability, and adaptability without impairing system performance.

Furthermore, the linear searching method inside this storage is very time-consuming as the different replica copies are located in different regions. Searching in object storage is not significant or efficient that way, since it stands. Content-Based Search (CoBS) primarily denotes the search that investigates the contents of inputted data rather than the metadata connected with the data, such as keywords, tags, or descriptions. In this usage, “content” may indicate colors, forms, materialistic details, or any other information obtained from the data itself. Manually annotating photos by inserting keywords or information into a huge database takes time and may not catch the keywords intended to identify the data.

The interest in CoBS is starting to grow as the usage of data is increasing and metadata-based systems are struggling to work on a large amount of data. Existing technology can rapidly search for information about any data, but this requires humans to manually characterize each image in the database. This can be difficult for extremely big databases or photos created automatically, such as those from surveillance cameras. Images that utilize various synonyms in their descriptions may also be overlooked. Systems based on classifying photos in semantic classes like “cat” as a subclass of “animal” may avoid the miscategorization problem, but will take more labor by a user to locate images that may be “cat”, but are only classed as an “animal”. Many standards for categorizing photos have been proposed, but all encounter scaling and miscategorization difficulties. Also, to our knowledge, there has been no work based on content searching across different types of information in one architecture.

In this Chapter, we propose Sherlock, a CoBS architecture for an object storage system that enables us to extract additional metadata from images and keywords from documents and store them in a metadata database that helps us search for our desired data based on its contents. In this Chapter, we are referring to content as the objects present in images and documents. In order to do so, we firstly identify the type of the file. Considering the file is image, we extract the information using an object detection Convolutional Neural Network (CNN) model named DarkNet, YOLOv4 [147] and YOLOv8 [148] architecture to detect objects. On the other hand, for document files, we extract the information using one of the Natural Language Processing (NLP) architecture, BERT [149]. Afterwards, we retrieve additional data such as the object path in the form of an HTTP link. The data is passed to an Elasticsearch Cluster (ESC) [150] and the object is uploaded to object storage systems like OpenStack Swift. When the user searches for an object, our proposed interface takes input from the user, performs a search in the ESC, and returns a list of objects. The user can be able to access the objects from Swift. In this way, there is only one GET request to the object storage system. Besides, the enriched content metadata are created using BERT and DarkNet and stored in ESC ensuring more relevant content searching for the user. The basic objective of our study is as followed:

1. To find an alternative method of content (documents and images) search for object storage systems like OpenStack Swift.
2. How does our alternate method perform in an average computational environment?
3. How can it be used as a user-centered image retrieval system?
4. How much delay can occur depending on the size of the images when uploaded as significant batches?
5. How does Elasticsearch respond to massive amounts of images?

We propose the following contributions to this Chapter based on our findings -

- Our work is the first to come up with an architecture that can jointly do CoBS inside images and documents.

- We create the OpenStack Swift JOSS client User Interface (UI) in order to access Swift and the Elasticsearch cluster at the same time using user-level authentication tokens.
- We rigorously test our BERT, DarkNet, YOLOv4 and YOLOv8 algorithm with different custom-weighted files to get maximum accuracy. We use three different datasets and calculate the response time of the YOLOv4 and YOLOv8 object detector as well as its precision in detecting multiple objects in a single image.
- We use a pre-trained BERT model to extract keywords from Documents. Besides, in the Elasticsearch cluster, we perform multiple query requests from our User Interface to get the elastic cluster's response time and the average query time. Lastly, we add different filters to the search engine using the elastic cloud API.

We structure rest of our study as followed: Firstly, we look into some of the recent state-of-the-art research studies on different types of searching, extraction, and retrieval system in Section 5.2. Afterward, we investigate the backgrounds of the architectures that we use to make our whole study effective in Section 5.3. Then, in Section 5.4, we illustrate the methodology with implementation, which uses different cloud and deep learning-based technologies. Then, in Section 5.5, we evaluate the proposed architecture. After that, we discuss the evaluation of our experimental results and do a comparative study with some related works in Section 5.6. Finally, in Section 5.7, we discuss and demonstrate possible future works.

5.2 Related Work

The concept of searching in Object storage is not entirely new to us. Platforms like Amazon S3, and OpenStack Swift- all have their own kind of searching approaches. Although there has been very little research related to searching in object storages they have not been implemented on platforms like OpenStack Swift and other object storages. As a result, we study these research papers in order to understand their work and the complications they faced while working with object storage. We segment the search into two different categories. The first part aims to review previous relevant works in the field of searching in object storage, specifically different types of metadata-related searches. The second part emphasizes on Query related searches in Object storage.

5.2.1 Metadata Searching

Leung et al. [151] suggest a scalable index-based file metadata search system that outperforms competing solutions in terms of query performance while using less disk space, [151] named "The SpyGlass". The type of programs that operate with millions of data generate need to be analyzed petabytes of data which are divided into millions of files or objects, according to Singh et al. [152]. Additionally, they must maintain their metadata, which exponentially increases the total amount of data in the object storage. There are many types of metadata, ranging from their date of creation to size. So, they propose a Metadata Catalog Service (MCS) which can store and access different types of metadata, and users can query for any type of metadata they want.

In 2015, the creators of the OpenStack Swift platform introduce a new type of metadata searching at a Summit in Tokyo. In this Chapter, a metadata enrichment engine is proposed which extracts additional metadata from the objects and adds them on top of the already available metadata. Moreover, this metadata is indexed using the 'Indexer Middleware' and RabbitMQ [153] in the Elasticsearch cluster [150]. So whenever a search request comes in, the indexer middleware will intercept it and forward it to the Elasticsearch cluster where it can be searched and the search result will be provided using the same path. As a result of the metadata enrichment engine, the metadata of the object store will be far better and will be able to perform data analytics far more precisely on the object storage than ever before. Using the Elasticsearch engine they are able to get desired results in no time. A massive amount of data and metadata can be indexed properly using RabbitMQ. But this project has not been able to provide a standardized search API. As a result, it still depends on the PUT GET methods of the original OpenStack Swift. However, while adding extra metadata manually we also need to provide the already provided metadata too. Otherwise, the previous metadata will be overwritten.

5.2.2 Query Searching

Searching in object storage is now common in cloud systems. Through our studies and findings, we try to find out the drawbacks, and issues of searching, and how they can be solved. A study [171] describes that Swift is a proxy server-based design that has the scale-ability of clusters. From this study, we get to know that it has a common performance issue in context-switching and that it buffers when messages are copied for each feedback transfer. So, basically, they propose a change in Swift's

Table 5.1: Findings from literature review

Reference	Purpose of the study	Technologies used	Findings
Ren et al., 2023. [154]	Image Retrieving from large amount of image pool	Zero-shot, NormSoftmax loss,	Challenges and solution for image retrieving from sketch-based images
Kakizaki et al., 2023. [155]	Defense approach for deep neural network (DNN) based content-based image retrieval (CBIR)	Interval bound propagation (IBP), DNN	Attack defense against content-based image retrieving
Wang et al., 2023. [156]	Content based image retrieving in industrial section	CNN, Binarization, Pre-trained Alexnet	Deep learning to retrieve image with feature extraction and use of binarization in CBIR
Choe et al., 2022. [157]	deep learning based image retrieving	CNN	CNN based image retrieving improved accuracy than before
Monowar et al., 2022. [158]	New AutoRet content-based image retrieval system	Deep CNN, AutoEmbedder framework	Functionality of a CBIR system with spatial pooling, DCNN
Keisham et al., 2022. [159]	Search and Rescue based CBIR approach	SAR (search and rescue), Deep Nural Network-SAR (DNN-SAR)	Deep dive into DNN-SAR based CBIR workflows and positive outcome on image retrieving
Noor et al., 2022. [1]	Workflow of image retrieving in cloud system	Openstack , Image compression	Image retrieving technique in cloud.
Ahmadvand et al., 2021. [160]	Big data processing	DVFS (dynamic voltage and frequency scaling)	Big data processing technique using DVFS in variety of data
Xue et al., 2020. [161]	How mass metadata can be kept without reducing performance and make better transition in files retrieving	Openstack, UCARP	Faster file retrieve and better performance
Lima et al., 2019. [162]	OpenStack Architecture	Openstack, PRISMA, Horizon, Neutron, Zaqar, Swift, Barbican	A complete view of openstack architecture.
Evans et al., 2018. [163]	Object Storage searching with indexing and metadata.	SQL, Clueso, Amazon Athena, Cassandra	A few more options for searching metadata
Gugnani et al., 2017. [164]	Find out the common issues of Swift default architecture and designs and new implements.	InfiniBand, RoCE such as RDMA	Workload increases up to 2x and performance boosted up to 7.3x
Noor et al., 2017 [53]	Reliable and secure image processing	P-Fibonacci transform of Discrete Cosine Coefficients(PFCC), Openstack Swift	Performance improvement of image processing and retrieving with the proposed framework
Yigal et al., 2016. [165]	Monitoring tools for OpenStack	Openstack, Elasticsearch, Logstash, Kibana	A way to monitor and retrieve Openstack's data with middleware tools
Biswas et al., 2015 [166]	ACL in OpenStack	Openstack Swift, Swift ACL, JSON	User-level access with ACL feature to retrieve the object.
Wang et al., 2015 [167]	Ciphertext based CBIR system	CKKS (Cheon-Kim-Kim-Song) homomorphic encryption, DNN	Using DNN and CKKS, LHS (locally sensitive hash) retrieved image from ciphertext
Raghuveer et al., 2007 [168]	How Intelligent storage works for data	Node based ISN OSD, Squad framework	Squad framework performs better than database-file system
Brandt et al., 2003 [169]	How to access large scale of metadata avoiding bottleneck	LH (Lazy Hybrid), OBSD, ACL	High performance cluster with scalability and flexibility.
LeCun et al., 1998 [170]	Character recognition using supervised learning	Graph transformer (GT), Optical character recognition (OCR), K-NN (K-nearest Neighbor), SDNN (space displacement neural network)	Hand fullness of GTN (Graph transformer network) in text retrieving

architecture that will provide much faster bandwidth with minimal latency while interacting with technology like RoCE, InfiniBand, and Remote Direct Memory Access (RDMA) [164]. They also propose other changes in Swift's operation design to improve its performance in transferring objects faster through the I/O module, which is based on RDMA. They also try an efficient hashing algorithm to make the verification faster in Swift. According to the study, they work on Swift bench-marking and find an improvement in managing workloads that is twice as fast as before, as well as performance boosts of up to 7.3×. According to Swift problem findings, the proxy-based architectures omits the best output can be produced and capacity of Swift clusters. They also make changes to the network. They find out the reasons for bottlenecks in the default Swift architecture and designs, implement a high-performance I/O framework for faster object transferring, and experiment with the use of various hashing algorithms to enhance object verification procedures.

Imran et al. [172] present some probable problems with metadata-related issues that we may face. In cloud storage, a lot of metadata are created which hampers the performance of the system. They propose an optimized solution for storing massive metadata which has improved load balancing modules and merges storage facility. Xue et al. [161] use HAproxy and UCARP to handle when huge amounts of metadata and it also reduces the buffering and accelerates read and write performances and overall throughput. Metadata is basically stored in a system as small files and with the increasing use of automated technology and remote sensing technology, lots of metadata is produced every day. As from the study, the traditional ways of retrieving metadata from relational databases become an issue for performance and hard to scale and maintain. On the other hand, storage performance reduces when lots of small files are stored. They use OpenStack Swift and optimize its frameworks and as for the load balancing module they used HAproxy and UCARP technology to overcome the bottleneck in the storage server. To make the performance better they transformed small files into large files so that small files cannot hamper the performance.

Biswas et al. [166] show how Access Control List (ACL) maintains accessibility and data security for all users. With ACL, it can be described who to give access to or not. On the other hand, they use JSON-based data to be sorted and how to protect data with JSON with labeling access control. In Swift, ACL and data security keystone has been used and stated how user data is checked through keystone and label user for granted for access data. For these, they change inside of the object server and its storing policies. Each time a request is made for downloading data it is checked by ACL, if it has been granted access or not. If it has, then it sends a request to read data from a server through a proxy server, otherwise, it stops it when it finds out for no ACL access. As for the storing policies they made, two types of policies for two types of data. One is LaBAC for user label data and objects label values, another is content level for JSON paths and labels. They find a drawback to their work, stating as it can only work with objects with applications or in JSON. Objects without a JSON file create issues with sending the full content of the files without requesting.

5.2.3 Content-based Image Retrieval System

Ren et al. [154] propose an Approaching-and-Centralizing Network (termed "ACNet") to jointly optimize sketch-to-photo synthesis and the image retrieval where the retrieval module guides the synthesis module to generate large amounts of diverse photo-like images which gradually approach the photo domain. Here, these diverse images generated with retrieval guidance can effectively reduce the overfitting problem troubling concrete category-specific training samples with high gradients.

Kakizaki et al. [155] introduce a defense approach for deep neural network (DNN) based content based image retrieval (CBIR) against adversarial examples (AXs). They first define newly certified robustness of CBIR, which guarantees that no AX that changes the ranking of CBIR exists around the query or candidate images. Then, they propose computationally tractable verification algorithms that verify whether the certified robustness of CBIR is achieved by utilizing upper and lower bounds of distances between feature representations of perturbed and non-perturbed images. Afterward, they propose new objective functions for training feature extraction DNNs that increase the number of inputs that satisfy the certified robustness of CBIR by tightening the upper and lower bounds.

According to Wang et al. [167], the industrial CBIR as well as second processes: first, the task consists of two processes, firstly images are handled to the file server and recorded during the collection process, and secondly, the indexing step consists of three steps: feature processing and feature indexing, also feature extraction—relationships between the queries and images, including contents of collected images and computing them during the search step.

LeCun et al. [170] have accomplished the first excellent outcomes in character recognition using supervised back-propagation networks. Machine learning and image processing research are getting quite popular, and the operations of CNN have recently skyrocketed to a massive reduction in computer hardware, especially Graphics processing units. Krizhevsky et al. [173] present an eight-laayered CNN model, which win the championship for the analysis task of ImageNet's large-scale visual recognition challenge in 2012 (ILSVRC-2012). VGG [174] increases the depth of the convolutional network to 19 layers and wins first and runners-up place in the ILSVRC-2014 localization and classification. He et al. [175] present an approach by which they outperform GoogLeNet by 26 percent when studying rectifier neural networks. Choe et al. [157] propose a convolutional neural network based CBIR approach to diagnosing Interstitial Lung Disease with Chest CT. Monowar et al. [158] introduce AutoRet, a deep convolutional neural network (DCNN) based self-supervised image retrieval system. The system can work in self-supervision and can also be trained on a partially labeled dataset. The overall strategy includes a DCNN that extracts embeddings from multiple patches of images. Further, the embeddings are fused for quality information used for the image retrieval process.

Keisham et al. [159] present a Deep Search and Rescue (SAR) Algorithm-based CBIR approach. The steps involved in the proposed Deep Neural Network-SAR (DNN-SAR) are pre-processing, multiple feature extraction, feature fusion, clustering, and classification. Initially, Fast Average Peer Group (FAPG) filter is used to remove the noise in the pre-processing stage. Then multiple features like color, shape, and texture are extracted and feature vectors are calculated. All these three features are fused into a single feature using average and weighted average techniques. Next, the fused features are clustered using the adaptive Sunflower optimization (SFO) algorithm. Finally, the relevant images are retrieved using DNN-SAR optimization algorithm.

Schall et al. [176] come up with a protocol for testing deep learning based models for their general-purpose retrieval qualities. After analyzing the currently existing and commonly used evaluation datasets they conclude with the result that none of the available test sets are suitable for the desired purpose and present the GPR1200 (General Purpose Retrieval) test set. Unlike the existing datasets, this dataset focuses on high domain diversity to test retrieval systems for their generalization ability. Images were manually selected to ensure solvability and exclude overlaps between categories of different domains.

Wang et al. [156] propose a secure and efficient ciphertext image retrieval scheme based on image content retrieval (CBIR) and approximate homomorphic encryption (HE). First, they use approximate homomorphic encryption to encrypt images after resizing and uploading the ciphertext images to the cloud for feature extraction of ciphertext. At the same time, the large images (size, dimension, and resolution) will generate data inflation after using homomorphic encryption. Therefore, the original images are encrypted using the chaotic image encryption scheme to reduce ciphertext size and computation costs. Second, they propose two deepening network depth optimization strategies that address the problem of insufficient neural network depth. Finally, reducing the dimensionality of the ciphertext feature vector using locally sensitive hashing (LSH) can accelerate the retrieval of ciphertext images.

Ahmadvand et al. [177] develop a data variety aware approximation approach, named Gapprox. They use a certain type of cluster sampling to improve the accuracy of data variety estimation. They divide the input data into some blocks considering the intra/inter cluster variance. The size of the block and the sample size are determined in such a way that by processing small amount of input data, an acceptable confidence interval and error bound is achieved. Then, the researchers use a data-variety-aware resource allocation approach [178] to reduce the processing cost of the considered job. For this issue, they divide the input data into some data blocks. They define the "significance" of each data block and based on it we choose the appropriate VMs to reduce the cost. For detecting the significance of each data portion, they use a specific sampling method. This approach is applicable to accumulative applications. They also [160] use Dynamic Voltage and Frequency Scaling (DVFS) architecture to reduce the energy consumption of computation to work with Data variety. Afterwards, they present an updated approach [179], named SAIR, to improve Quality of Result (QoR) of big data processing for budget-constrained aggregative usages based on significance variety.

Noor et al. [1] propose a novel approach to retrieve images faster by customizing the attributes in bit pixels of distinct luma and chroma components (Y, Cb, and Cr) of progressive JPEG images. Furthermore, new lossy PJPEG users of architecture to reduce the file size as a solution to overcome the possible drawback of this change proposed by them. Their proposed orchestration is reported to have a better response time from users up to 54% and a decreased image size of close to 27%. Moreover, this approach ensures up to 69% faster loading times. However, they only focus on image retrieval, which does not focus specifically on content.

In most recognition tasks, CNN-extracted features are considered the primary candidates. Razavian et al. use [180] to measure CNN performance, implement the OverFeatand perform network as a feature decoder on a variety of detection tasks and provide a platform for various datasets. According to Koskela et al. [181], CNN has been prepared on various object identification datasets can extract features for scene recognition tasks. These findings indicate that feature mining is possible, compelling, and extremely positive with CNN.

5.2.4 Keyword Extraction from Document

Researchers [182] present a multimodal key-phrase extraction approach, namely Phraseformer, using transformer and graph embedding techniques. Xiong et al. [183] propose Semantic Clustering TextRank (SCTR), a semantic clustering news keyword extraction algorithm based on TextRank that uses BERT to perform k-means clustering to represent semantic clustering. Then, the clustering results are used to construct a TextRank weight transfer probability matrix. Finally, Iterative calculation of word graphs and extraction of keywords is performed.

The recent solutions for searching in object storage and their findings are presented in Table 5.1. Their drawbacks inspire us to come up with a new solution with the help of object detection and natural language processing that is robust. Previous solutions do not have content-based data extracting and searching options and also do not have a floor to optimize the searching mechanism for best interests and also the integration of third-party search engine adaptation. To the best of our knowledge, our proposed methodology is the first to focus on these aspects.

5.3 Background

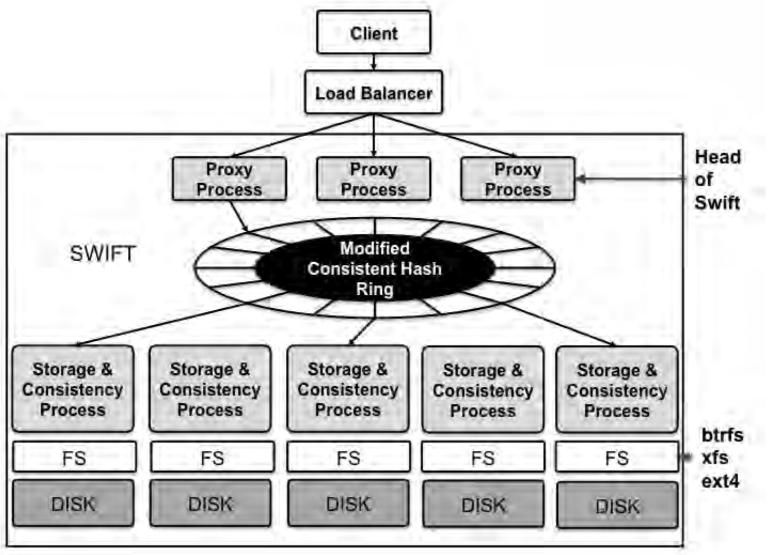

Client
Load Balancer
Proxy Process
Proxy Process
Proxy Process
Head of Swift
SWIFT
Modified Consistent Hash Ring
Storage & Consistency Process
Storage & Consistency Process
Storage & Consistency Process
Storage & Consistency Process
Storage & Consistency Process
FS
FS
FS
FS
FS
btrfs
xfs
ext4
DISK
DISK
DISK
DISK
DISK

Figure 5.1: Overview of the storage architecture [3]

This section goes over the fundamental architectural framework of Swift, YOLO, BERT, and Elasticsearch.

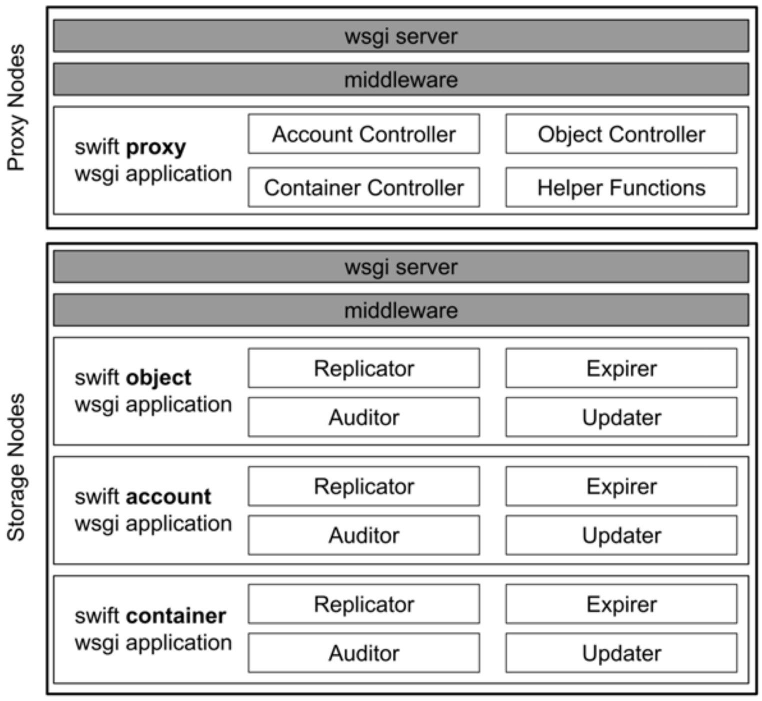


Figure 5.2: Different consistency processes and layers in proxy and storage nodes of OpenStack Swift

5.3.1 Architectural Overview of Swift

OpenStack Swift is a highly scalable object storage that is designed keeping in mind the phrase that “Failure is a common occurrence”. As a result, Swift is divided into 4 subsections: Proxy, Account, Container, and Object nodes. A proxy server is located in the first layer. Data that goes in and out of the storage has to go through the HTTP file transfer protocol. The requests for data are done by API requests. The task of the proxy server is to capture the requests and work accordingly. The proxy server determines the location of the data or its storage node by the URL. There are Rings, which keep the address of the information like names and entries that are stored on the cluster. It also keeps track of the path of the data. The way Rings keep the mapping work is by introducing zones, devices, partitions, and replicas. Zones may be any storage device like a hard drive to a full server. After that, there are containers and accounts. The list of containers in a particular account is stored in that account’s database. Swift has multiple object nodes which are independent of each other. These object nodes are easily replaceable in the event of any failure. However, Swift has an internal replication system that replicates the stored object into a minimum of three different nodes. So when one node is replaced the objects are not lost [162]. Figure 5.1 presents the architectural overview of OpenStack Swift and Figure 5.2 presents the different consistency processes and layers in proxy and storage nodes of OpenStack Swift.

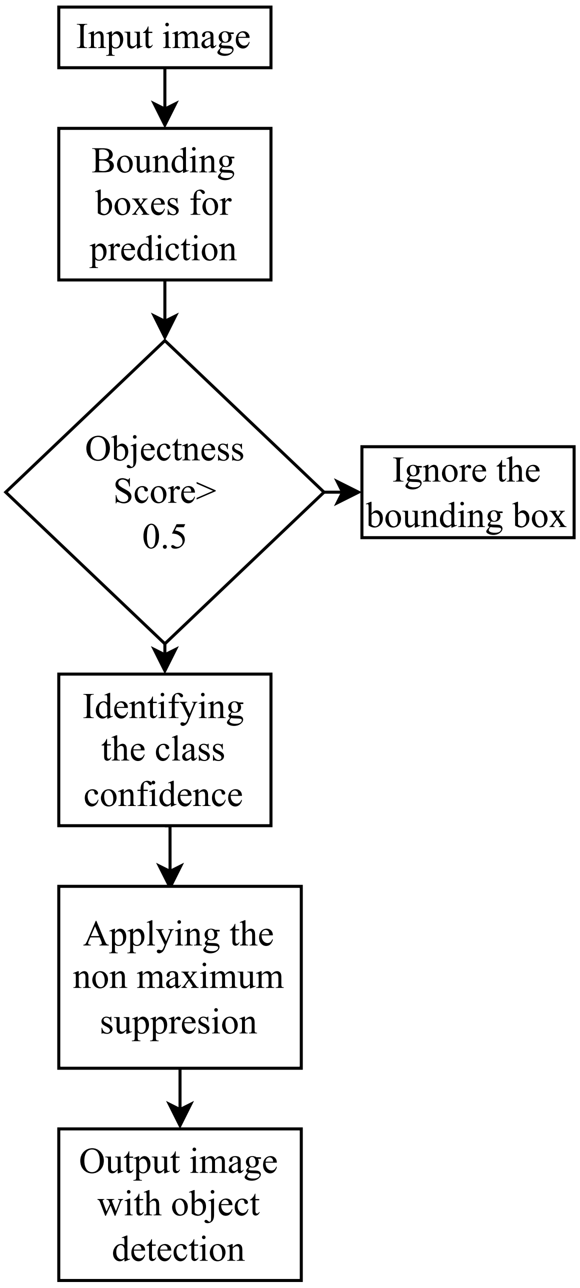

Input image
Bounding boxes for prediction
Objectness Score> 0.5
Ignore the bounding box
Identifying the class confidence
Applying the non maximum suppresion
Output image with object detection

Figure 5.3: Workflow of YOLOv4

5.3.2 YOLOv4

YOLOv4 [184], *You Only Look Once* Version 4, is a sophisticated technique for single-stage object detection that utilizes regression techniques to gain a good precision score and can be performed concurrently, It is the predecessor of the YOLO line of algorithms. This algorithm is introduced in 2020 by Bochkovskiy et al. [184] and builds upon the capabilities of previous versions such as YOLOv1, YOLOv2, and YOLOv3., achieving the finest potential balance between detection efficiency and precision at this time. Its architecture, which includes of the backbone, the neck, and the prediction, is shown in Figure 5.3.

An identical YOLO head used in YOLOv3 is applied in YOLOv4 for detection with three levels of granularity and anchor-based detecting steps [184]. The backbone of YOLOv3 combines the ResNet structure with a residual module to create Darknet53 [185]. Essentially, an object detector's backbone network undergoes pretraining on ImageNet classification. Pretraining alludes to the network's weights being modified for the new task of object identification even when they have already been trained to produce better feature extraction in an image.

The authors suggested the following backbones for the YOLOv4 object detector: CSPResNext50, CSPDarknet53, and EfficientNet-53 [184]. As for our purpose, we use CSPDarknet53. YOLOv4 resembles YOLOv3 in this way but adding Cross-Stage Partial Network's greater learning capacity (CSPNet) on Darknet53 emerges as a new addition and efficiency in YOLOv4 [186]. In YOLOv4, the CSPDarkNet53 architecture is introduced. This architecture includes a residual module where the feature layer is re-entered, resulting in increased feature information. This entitles the model to learn the distinction between output and input. In the residual module, enabling residual learning while also reducing the model's parameters and improving feature extraction [184]. The SPPNet and PANet can be merged to form the neck of the model. The SPPNet involves designing and simulating the feature layer multiple times before applying maximal pooling with a large number of pooling cores of various sizes to the input feature layer [184]. The aggregated results are then concatenated and fused multiple times to strengthen the network's receptive field. Using the operations of Backbone and SPPNet, PANet convolves the feature layers. Then it up-samples them, doubling the height and width of the original feature layers, and then concatenates the feature layers generated after convolution and up-sampling with the feature layers obtained by CSPDarkNet53 to achieve feature fusion [184], and then down-sampling, compressing the height and width, and finally stacking with the previous feature layers to realize 5× more feature fusion. The Prediction module can bring predictions based on the network features. Using a 13 × 13 grid, for example, is equivalent to dividing the input image into 13 × 13 grids, with every grid being pre-set with three earlier frames. The network's prediction results will shift the placements of the three previous frames, and finally, it will be filtered by the non-maximum suppression (NMS) [187] algorithm to obtain the final forecast frame.

YOLOv4 introduces a novel panoramic data augmentation strategy to boost the dataset and use Complete Intersection over Union (CIoU) as the precision loss function, which make the network more probable to optimize in the direction of enhancing overlapping areas, thus stimulating the accuracy [188].

CloU is the loss function that combines classification loss, confidence loss, and complete intersection over the union. Up until the Intersection over Union (IoU) loss calculates the overlap between the expected and true bounding boxes. The CIoU loss eliminates this by introducing new terms that penalize differences in the aspect ratio and center point of the predicted and ground truth bounding boxes. This produces more precise findings and eliminates errors in localization and misclassification.

5.3.3 YOLOv8

YOLOv8 is the successor of all the previous YOLO models presented. It is introduced by Jocher et al. [148]. YOLOv8 has an architecture similar to one of its ancestors, YOLOv5. It does not have a Darknet backbone like its' predecessors. It is based on PyTorch. And it has a Python backbone rather than Darknet, which is based on C used in YOLOv4. This is convenient for the users to make it customizable and bring improvements through the model. These inspire researchers and engineers to create new models based on that. There are three main aspects of the YOLOv8 model. Firstly, this model is anchored free and it does the prediction from the center of the object and has the attributes of an anchor-free model. Then, the architecture of the convolutional kernel has been changed from 1×1 to 3×3, and lastly, it uses mosaic augmentation at the time of training which is quite a praise able work [189]. As a classification loss, YOLOv8 employs DFL (Distribution Focal Loss) Loss + CIoU Loss and VFL (Vertical Flipped Label) Loss. VFL loss flips the labels of good examples. The label is substituted by its vertical inverse for each positive example, resulting in a new negative example. It solves the localization problem and also adds other confidences and classification losses to measure the weight [190]. On the MS COCO dataset, YOLOv8x achieve an AP of 53.9% with a 640-pixel image size (compared to 50.7% for YOLOv5 on the same input size) at a speed of 280 FPS on an NVIDIA A100 and TensorRT [191].

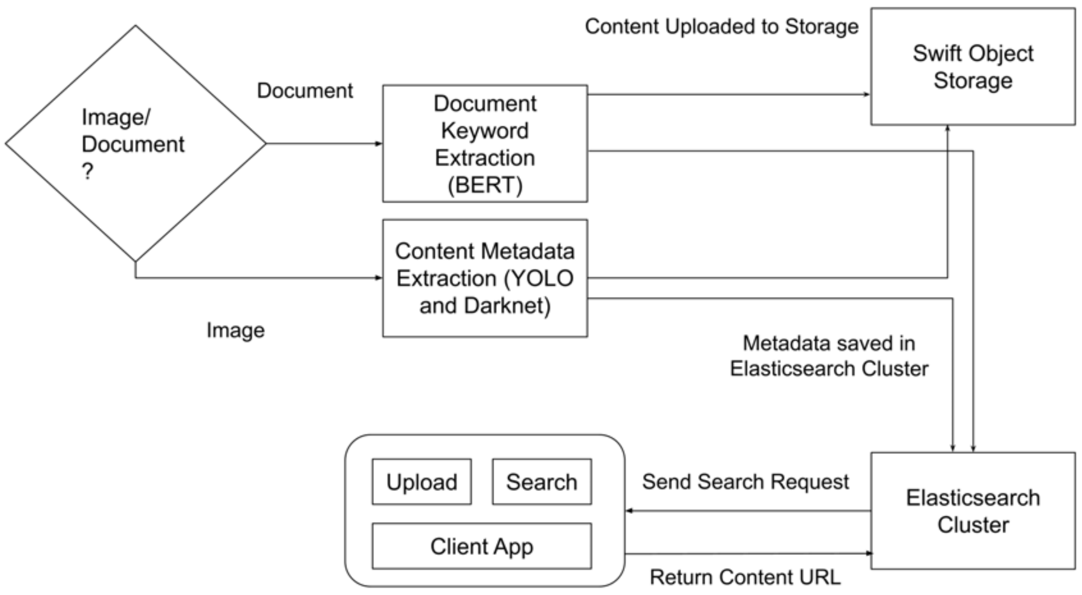

Image/ Document ?
Document
Image
Document Keyword Extraction (BERT)
Content Metadata Extraction (YOLO and Darknet)
Content Uploaded to Storage
Swift Object Storage
Metadata saved in Elasticsearch Cluster
Upload
Search
Client App
Send Search Request
Return Content URL
Elasticsearch Cluster

Figure 5.4: Overview of the proposed architecture

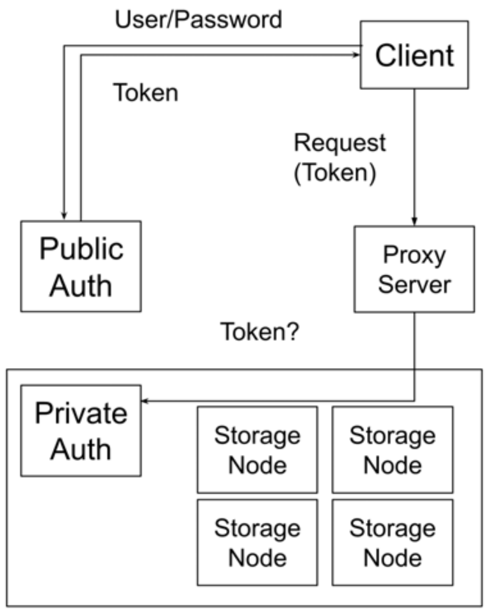


Figure 5.5: Java library for OpenStack Swift (JOSS)

5.3.4 BERT

The BERT (Bidirectional Encoder Representations from Transformers) model architecture is based on a multi-layer Transformer encoder, which was originally implemented by Vaswani et al. [192]. Devlin et al. [149] introduce the BERT Transformer based on using bidirectional self-attention. This bidirectional mechanism removes the restrictions that self-attention can only incorporate the context from one side: left or right. Different from other embedding generation architecture, such as Word2Vec [193], the input to the BERT model is not vectors that represent words. Instead, the input includes token, segment, and position embeddings. The token embedding is WordPiece embeddings [194] that contain 30,000 tokens. The base BERT model is pre-trained using two unsupervised tasks:

1. Masked Language Model (LM) - a task to predict some random masked tokens in the input. The objective is to train a bidirectional encoder.

2. Next Sentence Prediction (NSP) - a task to predict the following sentence of the input sentence.

The objective is to understand sentence relationships so that the pre-trained BERT model can be a better fit for other NLP applications, such as Question Answering (QA) and Natural Language Inference (NLI) where sentence relationships are crucial. In this research, we make use of the base BERT model that is in TensorFlow Hub. It has 12 transformer blocks, 12 self-attention heads, and a hidden size of 768. The BERT base model can be fine-tuned for text classification by simply adding a softmax classification layer on top of the BERT model to predict the class c of a given text sequence, as Equation 5.1. The input to the softmax layer is the last hidden layer output H of the first token that represents the original text sequence.

$$p(c \mid H) = \text{softmax}(WH) \tag{5.1}$$

Here, W are the parameters for the classification layer. They are fine-tuned with all the parameters from BERT to maximize the log probability of the correct label. Even though The BERT model is large and slow to train due to its complex structure and extensive corpus, but its advantages include increased accuracy, better generalization, and the capacity to apply acquired information to different tasks. Additionally, methods and improvements, such as model pruning, knowledge distillation, and hardware acceleration, can be utilized to lower the computational cost of developing and utilizing the model.

5.3.5 ElasticSearch Overview

Elasticsearch is a full-text search library based on the open-source search engine Apache Lucene. It is capable of performing a full-text search. It can conduct a structured search, analytics, or a combination of all three as this is built for real-time, distributed search and data analysis [195]. The highly adaptable query API of Elasticsearch allows for the simultaneous use of filtering, sorting, pagination, and aggregation in a single query [150]. Elasticsearch is capable of easily handling unstructured data, and allowing for the indexing of JSON files without the need for a prior schema. It automatically attempts to identify class mappings and adjusts for new or removed fields. It also offers built-in functionality for clustering, data replication, and instantaneous fail-over, all of which are transparent to the user [150].

On the other hand, Elasticsearch is a data store with fast and powerful search capabilities, real-time indexing and analytics, and flexible data modeling capabilities. It is best suited for full-text search and analytics use cases, Hadoop for distributed computing and large-scale data processing, and MongoDB for document-oriented data storage and retrieval needs.

5.4 System Design and Implementation

Figure 5.4 presents the methodology of our proposed system. First, it checks if the file is an image or a document. Based on the file type, the content is sent to extract the crucial information. For images, the content is sent to extract the metadata. And for documents, the content is inputted to extract the keyword. Darknet (for image) and BERT (for document) process the metadata and send the data to the Elasticsearch cluster. The object detection/keyword extraction is done on the client app. When the metadata is uploaded in the Elasticsearch cluster, the content is uploaded to the Swift server.

5.4.1 Developing Client-side

We use Java client for OpenStack Swift (JOSS) [196], in Figure 5.5, to build the client app. We use Elasticsearch as it offers multi-language support for handling request and response data, language detection libraries, and plugins and integrations to provide additional language-specific functionality. And our JOSS client connects well with Elasticsearch. The location path of the content in our storage

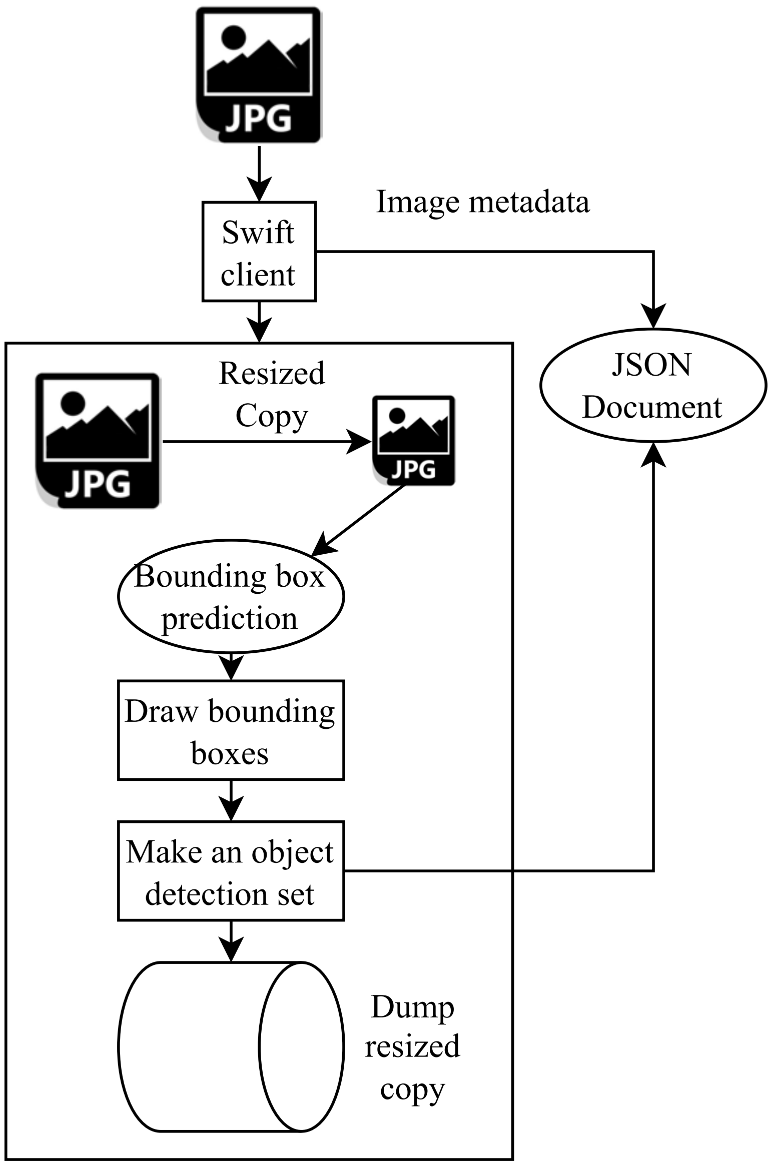


Figure 5.6: Overview of the modified YOLOv4 and YOLOv8 object detectors

server is saved in the Elasticsearch cluster. We use Elasticsearch as it offers multi-language support for handling request and response data, language detection libraries, and plugins and integrations to provide additional language-specific functionality. And our JOSS client connects well with Elasticsearch. When the user searches for content, the client app performs a search in the Elasticsearch cluster and returns content suggestions to the user with the Swift location path. This ensures minimum load on the Swift server and accurate searching based on the metadata. Because of this extraction, the proposed system has a sound knowledge of each content.

OpenStack has a few libraries to interact with the Swift object storage system [171]. We use the Java library for Openstack Swift (JOSS) [196] to develop our client app. It is a desktop-based application where we have our object detection model as well. Object detection is done using the client device's computational power. Then we upload the metadata data to the Elasticsearch cluster and the content to the Swift server. JOSS provides many features to interact with the Swift servers including authentication, object uploading, content location path generation, and so forth [197].

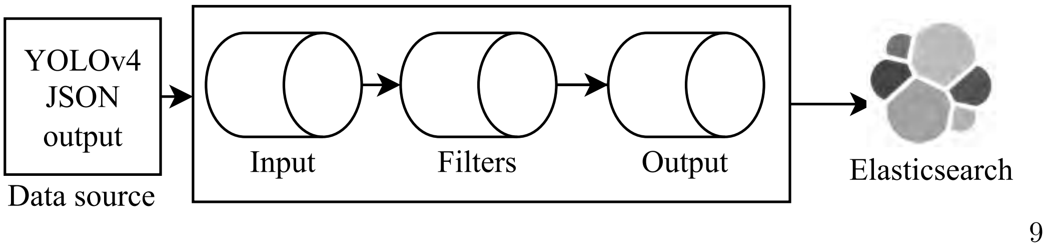


Figure 5.7: Workflow for JSON document in Elastic Cluster

5.4.2 Developing Keyword Extraction

BERT is one of the state-of-the-art models to solve problems related to Natural Language Processing (NLP) which uses Attention-based mechanisms. In our case, we take a document (docx/pdf) and then take all the text from that document and put it in the BERT model and we the get best five three-word keywords out of the document we input.

5.4.3 Developing Object Detection

Figure 5.6 shows how our proposed YOLOv4 algorithm works after it gets an image. YOLOv4 provides us with fast and accurate object detection with the help of bounding boxes and non-maximum suppression. However, in our case, we do not want an edited image with bounding boxes. As a result, we propose a different workflow for the YOLOv4 where after getting an image it will make a copy of that image and perform necessary detections. We also followed the same approach for the latest YOLOv8 too And we used a pre-trained YOLOv8 like we used a pre-trained model for YOLOv4 which is already trained with the MSCOCO-17 dataset which has 80 classes of data.

At that time, the actual image will be sent directly to the storage server. And after that, the detection is done the copied image will be dumped and the object detection set with other metadata related to the image including the object URL path will be written into a JSON document. Lastly, the JSON document will be pushed into the Elasticsearch server.

5.4.4 Developing the Storage System

Figure 5.1 shows the overall architecture of the storage system. After Detection is done and the object is pushed to Swift, it goes into Swift using the Swift proxy pipeline. In our model, we use a multi-node Swift setup. There are 3 proxy servers and multiple object servers. The load balancer chooses the suitable proxy server for the object. The proxy server sends the object to the Ring, from where the object is sent to the appropriate object server. In our model, we do not change how Swift handles these requests in order to maintain its scalability and compactness.

5.4.5 ElasticSearch Cluster

Figure 5.7 shows the workflow for JSON document in Elastic cluster. We set up an Elasticsearch server in a different Virtual Machine with a Logstash pipeline where the JSON file generated by the object detector gets pushed. Our Logstash pipeline filters out the unnecessary data from the JSON file such as the coordinates of the bounding boxes, class id. It also formats the JSON file in a way that is easier for Elasticsearch to index properly based on the image file name and the contents of the image which in our case are the detected objects.

5.5 Performance Evaluation

We measure the performance of our suggested architecture using a real-world implementation. Before this, we first elaborate on proposed experimental settings.

5.5.1 Experimental Setup

We set up multiple virtual machines using the Google Cloud Platform (GCP). One machine work as a proxy server and the other one as the object server node. Account and container servers are included in the object server machines which are shown in Figure 5.8. We use our local machine to detect and upload images for this testing. The configuration for our local machine is as follows: Intel(R) Core i5-7300HQ CPU having a 2.50 GHz base clock speed, 8 GB ram, and an Nvidia GTX 1050 Graphics Unit.

Next, we use another graphics unit Nvidia RTX 2070 where we run our testbed with YOLOv8.

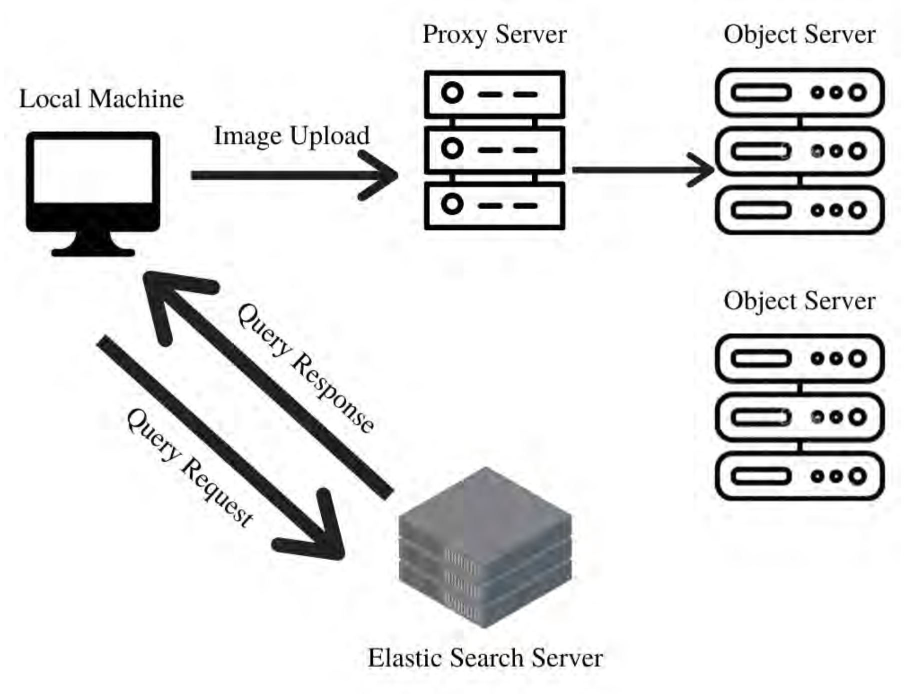


Figure 5.8: Experimental setup overview

Furthermore, we use another virtual machine as the Elasticsearch server, which is configured to get data dumps from our local machine. For all the Virtual machines, we use the same kind of setup: 4 GB ram, 60 GB storage, and the operating system is Ubuntu 18.04.

5.5.2 Dataset

We utilize the Microsoft COCO Dataset [198] for our object detection module. The dataset comprises a total of 26,000 images and 80 classes. To conduct our testing, we divide the dataset into three segments, containing 1,000, 5,000, and 20,000 images, respectively. The distribution of these segments can be observed in Figure 5.9 and Figure 5.10. To evaluate the images, we employ pre-trained YOLOv4 and YOLOv8m models. The class distribution of the MSCOCO image dataset is illustrated in Figure 5.9 and Figure 5.10.

We employ the SemEval2017 Dataset [199] for our keyword extraction task. This dataset is composed of paragraphs that have been extracted from 500 journal papers sourced from the domains of Computer Science, Material Sciences, and Physics, available on ScienceDirect.

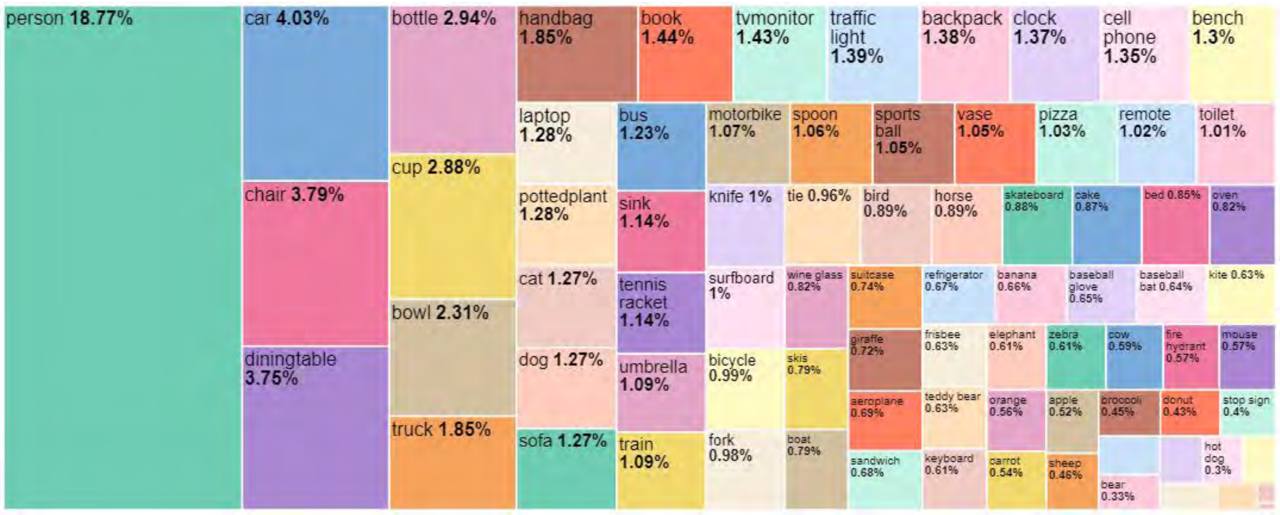


Figure 5.9: Class distribution of MSCOCO image dataset (number of images = 5000)

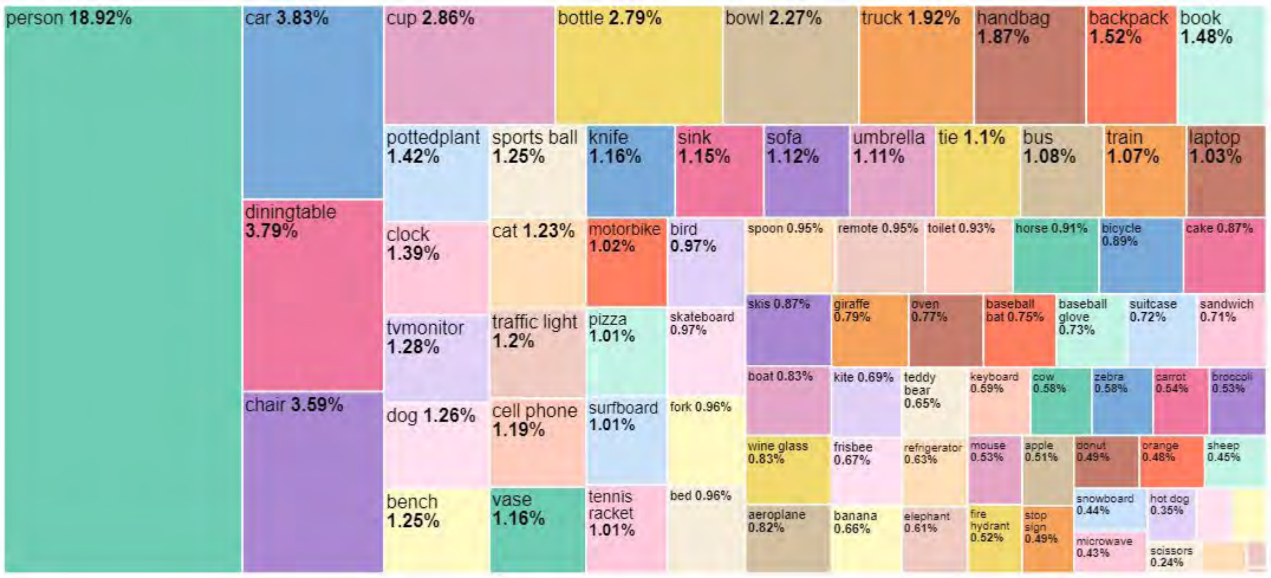
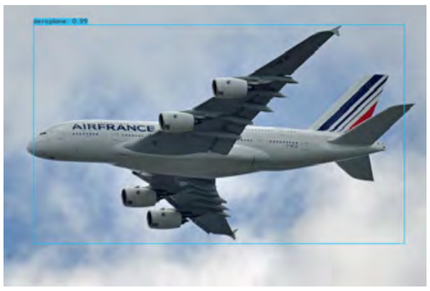


Figure 5.10: Class distribution of MSCOCO image dataset (number of images = 20000)

5.5.3 Experimental Results

In this section, we present the outcomes obtained from employing the datasets in our system. We begin by showcasing the results obtained from the image dataset, followed by the results from the document dataset.

(a) Aeroplane



(b) Aeroplane (Detected)

(c) Man on horse

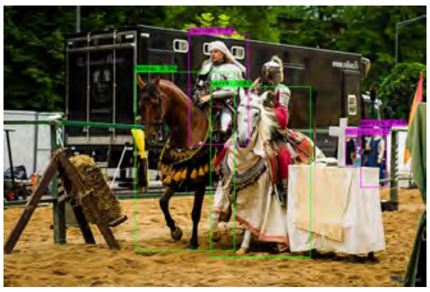

(d) Man on horse (Detected)

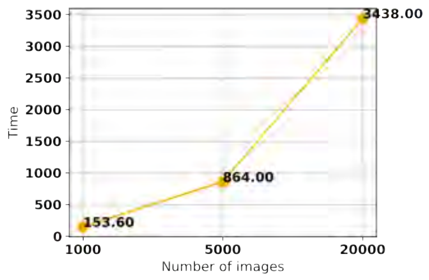
Figure 5.11: Object detection (single and multiple)

Table 5.2: Dataset testing metrics

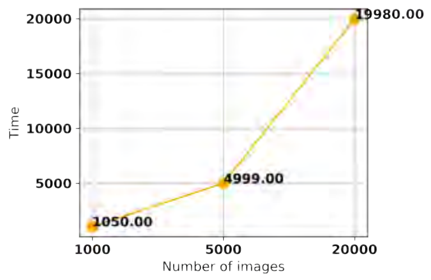
	5000 Images	20000 Images
Detection time (sec)	681	2907
mAP	0.71	0.73
Average IoU	0.55	0.56
F1 score	0.68	0.69
Recall	0.68	0.69
Precision	0.68	0.69

5.5.3.1 Image Dataset Test

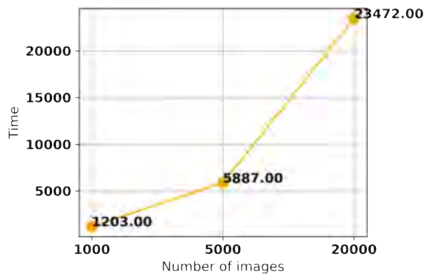
Table 5.2 displays various metrics obtained from our testing sets. The precision metric holds significance for our model as it indicates the system's ability to accurately provide users with the desired images. In our case, we achieved an mAP (Mean Average Precision) of 0.71 for the 5,000-image dataset and 0.73 for the 20,000-image dataset, resulting in an overall precision of 68.5%. Figure 5.11 illustrates the detection of both single and multiple objects by our model.



(a) Detection time graph (seconds)

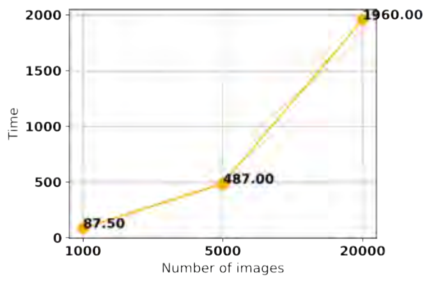


(b) Upload time graph (seconds)

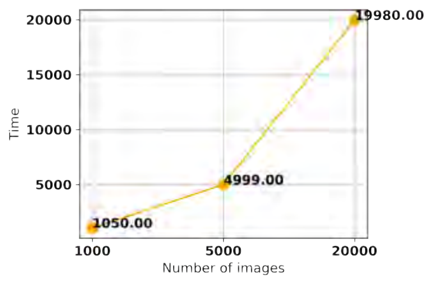


(c) Upload & detection time graph (seconds)

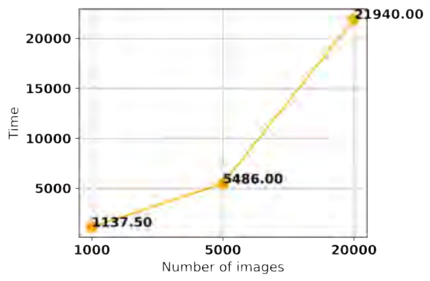
Figure 5.12: Time graph (YOLOv4)



(a) Detection time graph (seconds)



(b) Upload time graph (seconds)



(c) Upload & detection time graph (seconds)

Figure 5.13: Time graph (YOLOv8)

5.5.3.2 Detection Time Test

Figure 5.12a and Figure 5.13a depict the time taken to detect 1,000, 5,000, and 20,000 images. The curves in both graphs exhibit a slight upward slope, indicating that the detection time remains relatively low in comparison to the number of images. This efficiency is attributed to the YOLO algorithm, which lives up to its name by examining each image only once. Furthermore, the negligible impact on speed by removing the bounding box drawing method further supports the algorithm's efficiency.

5.5.3.3 Upload Time Test

During the uploading process, we impose a speed restriction of 2 Mbps (megabits per second) and compute the upload time for the images. Figure 5.12b and Figure 5.13b display a relatively steep curve, which is a result of the low upload speed in comparison to the file sizes of the images. This discrepancy leads to longer upload times, as depicted in the graphs.

5.5.3.4 Total Time for Proposed Model

Following the individual calculations of detection and upload time, we proceed to initialize our system to determine the combined time required for both uploading and detecting the various image sets. Figure 5.12c and Figure 5.13c illustrate a slight increase in the curve, indicating the cumulative time for uploading and detection.

5.5.3.5 Uploading and Detection Time Comparison

Based on Figure 5.14, we observe the comparison between the uploading time and the combined time for detection and upload. It indicates that the additional requirements of object detection and our system slightly increase the overall time needed to deliver the image to Swift. However, this difference is relatively insignificant considering the extensive work taking place behind the scenes.

It is important to note that our system utilizes an older graphical processing unit model, the Nvidia GTX 1050, without CUDNN functionality. As a result, the GPU usage reaches a maximum of 15-20%. Upgrading to a newer GPU model or using an older version with CUDNN enabled would significantly decrease the detection time, subsequently reducing the overall upload and detection time. This would bring the curves in Figure 5.14 much closer to each other.

Table 5.3: Average time (in seconds) to do different tasks for different data sizes with YOLOv4

Image count	Average detection time (YOLOv4)	Average upload time	Total upload & detection time (YOLOv4)
1000	153.6	1050	1204
5000	864	4999	5887
20000	3438	19980	23472

Table 5.4: Average time (in seconds) to do different tasks for different data sizes with YOLOv8

Image count	Average detection time (YOLOv8)	Average upload time	Total upload & detection time (YOLOv8)
1000	87.5	1050	1138
5000	443	4999	5442
20000	1790	19980	21770

Table 5.3 and Table 5.4 present the average time taken by our models to detect and upload various data segments.

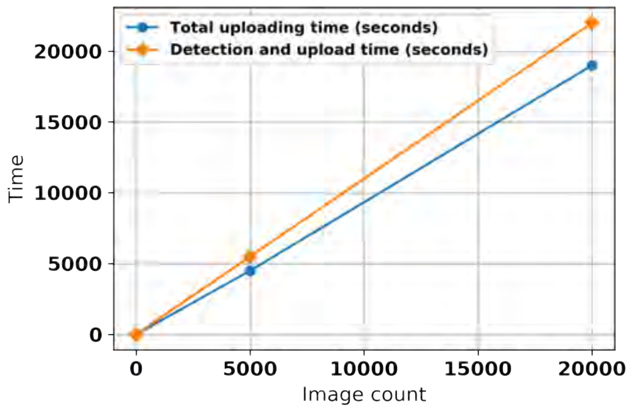


Figure 5.14: Comparison graph

5.5.3.6 Result Evaluation for Document

Table 5.5 presents the tentative document and the extracted keywords. We mention the exact documents in the appendix section. We find the documents which are most similar to the document and

Table 5.5: Extracted keywords from BERT

Document	Extracted Keyword 1	Extracted Keyword 2	Extracted Keyword 3
Document A	neural networks deep	including computer vision	deep neural networks
Document B	numerical insight recent	thermodynamic limit recent	theories complex boltzmann

these are those keywords that can best represent our document. We use cosine similarity between vectors and the document. We select only top three keywords from most similar candidates (documents) to the input document which has three words on each.

5.5.4 Search Analysis

Here, we discuss the functionalities of searching within our system.

5.5.4.1 Completion Suggester

The completion suggester is a valuable feature that offers auto-complete and search-as-you-type functionality within our system. It assists users by providing relevant results while they are typing, thereby enhancing the precision of their searches. Figure 5.15 demonstrates the completion suggester in action. To implement this feature, we utilized the Elasticsearch API, which we integrated with our Client API to generate completion suggestions for users.

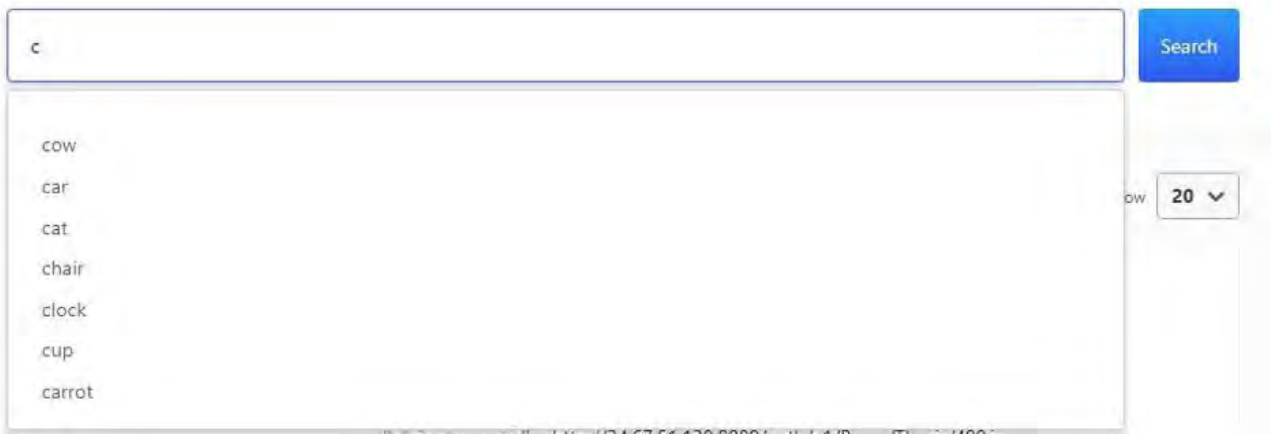
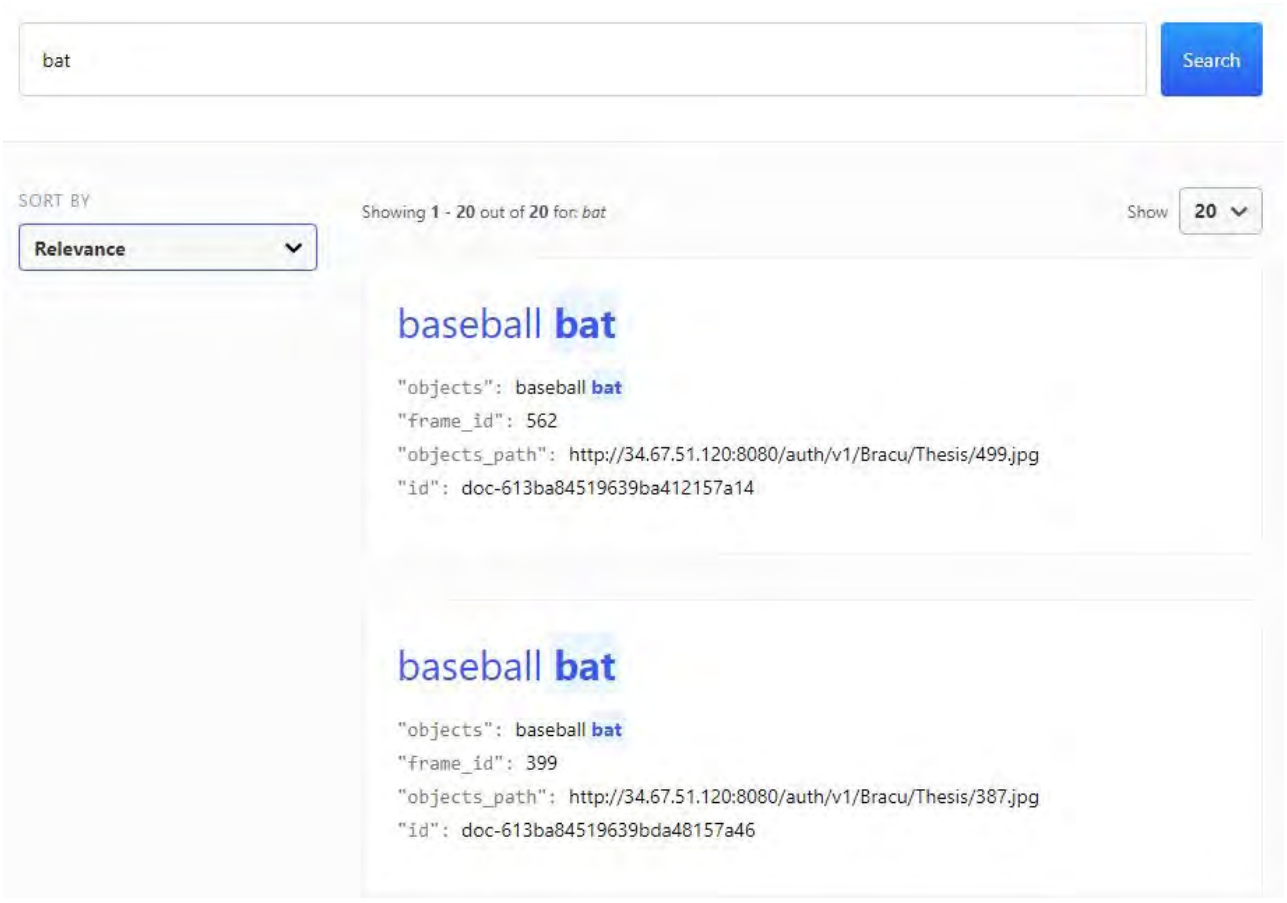


Figure 5.15: Our implemented completion suggester UI using Elasticsearch API in the backend

5.5.4.2 Search Based on Image Content

The Elasticsearch API serves as the interface to the Elasticsearch cluster, where all the documents are stored and indexed. In Figure 5.16, we can observe the search results based on detected objects. Notably, the "objects_path" field is visible, representing the address of the specific image in the Swift storage. This path encompasses the proxy address, account name, container name, and object name. In the given example, "http://34.67.51.120" denotes our proxy server, "Bracu" signifies the account name, "Thesis" represents the container name, and finally, "499.jpg" is the name of the object itself.



Showing **1 - 20** out of **20** for: *bat*

baseball **bat**

"objects": baseball **bat**
"frame_id": 562
"objects_path": http://34.67.51.120:8080/auth/v1/Bracu/Thesis/499.jpg
"id": doc-613ba84519639ba412157a14

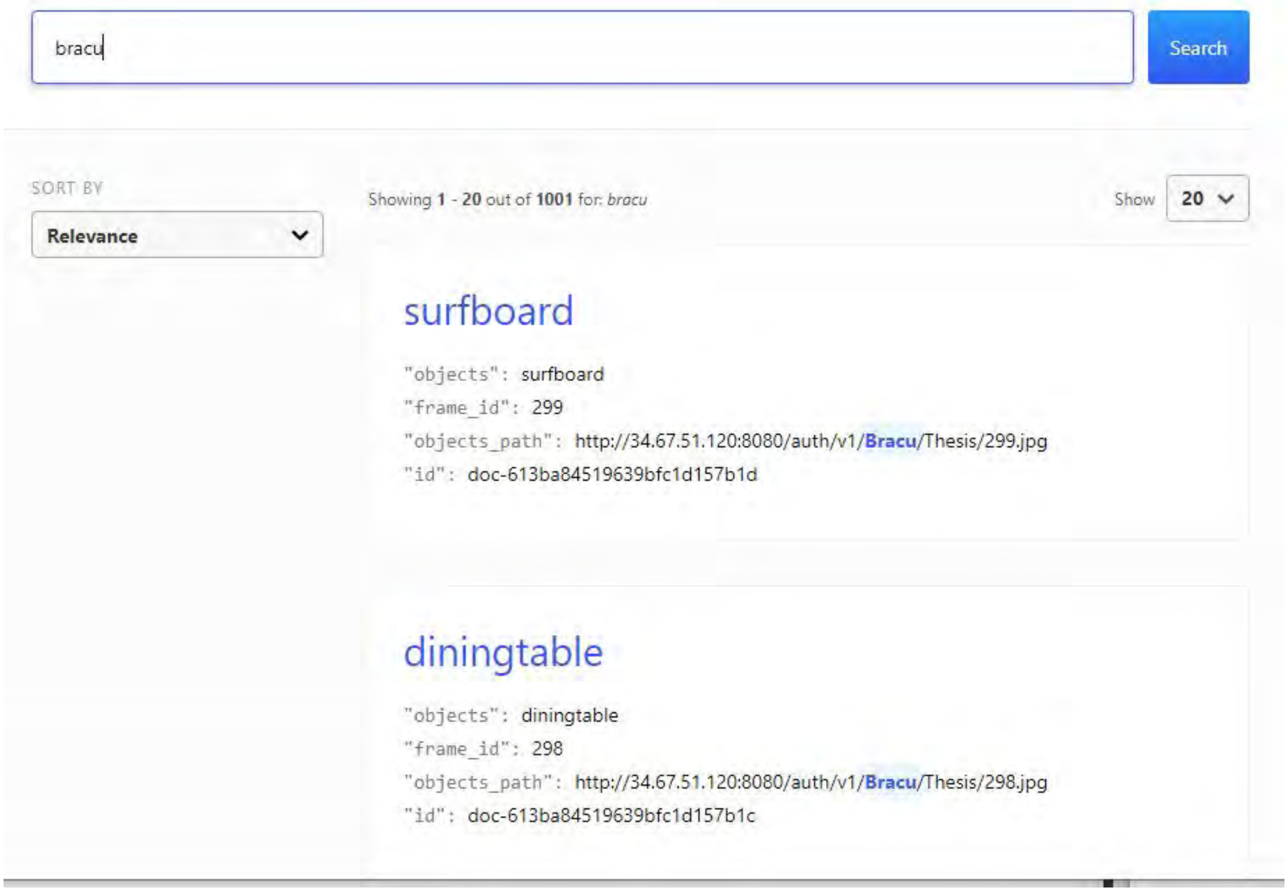
baseball **bat**

"objects": baseball **bat**
"frame_id": 399
"objects_path": http://34.67.51.120:8080/auth/v1/Bracu/Thesis/387.jpg
"id": doc-613ba84519639bda48157a46

Figure 5.16: Searching based on image content using Elasticsearch API

5.5.4.3 Search Based on Metadata

Figure 5.17 illustrates metadata-based searching within our system. In this example, "Bracu" represents our account name, which is one of the metadata fields collected and stored in the traditional Swift database. It is worth noting that other OpenStack platforms, like devstack and searchlight, utilize this type of metadata to perform searches within Swift. Our system demonstrates its capability not only to search based on image content but also to seamlessly conduct metadata-based searches.



Showing 1 - 20 out of 1001 for: *bracu*

surfboard

"objects": surfboard
"frame_id": 299
"objects_path": http://34.67.51.120:8080/auth/v1/Bracu/Thesis/299.jpg
"id": doc-613ba84519639bfc1d157b1d

diningtable

"objects": diningtable
"frame_id": 298
"objects_path": http://34.67.51.120:8080/auth/v1/Bracu/Thesis/298.jpg
"id": doc-613ba84519639bfc1d157b1c

Figure 5.17: Searching based on metadata using Elasticsearch API

5.5.4.4 Search Timing Results

In our dataset, we have a total of 80 classes. To evaluate the performance, we conducted a search using these 80 keywords and measured the average query time and average request time, as indicated in Table 5.6. Upon analysis, we observed that while the average request time displayed slight fluctuations, the average query time remained consistently low and stable.

Table 5.6: Average query time and request time for different data sizes

Documents	Total Keywords	Average Query Time(ms)	Average Request Time(ms)
1000	80	54	440
5000	80	57	570
20000	80	58	680

Table 5.7: Difficulty and availability of various implementations of Swift

Features	DevStack	Traditional Swift	SwiftStack	Proposed
Deployment	Developer	Developer	Commercial	Developer
Supported Operating Systems	Ubuntu, CentOS	Ubuntu, Fedora, CentOS	Windows, Ubuntu, CentOS	Windows, Ubuntu
Multi-Node Supports	No	Yes	Yes	Yes
OpenSource	Yes	Yes	No	Yes
Stability of Deployed Setup	No	Stable than Devstack	Yes	Yes
Difficulty to use	Low	High	Low	Low
Setup Difficulty	Low	High	Average	Average

Table 5.8: Comparison between various CBIR engines (here, TBIR = Texture Based Image Retrieval)

Author	Dataset	Images & number of classes	Techniques	Application	Query technique	Precision
Ashraf et al, 2020. [24]	Corel 1000	100 images with 10 classes	Video, Image data for content-based imagery collection via several methods	CBIR	Image	0.875
Ahmed et al, 2019. [25]	Corel 1000	1000 images with 10 classes	Image features information fusion	CBIR	Image	0.90
Nazir et al, 2018. [27]	Corel 1-K	1000 images with 10 classes	HSV analysis of colors, wavelength determination as well as the edge histogram characterization	CBIR	Image Text	0.735
Mistry et al, 2018. [200]	Wang	1000 images with 10 classes	Hybrid features and various distance metric	CBIR	Image	0.875
Liu et al, 2017. [201]	Brodatz	1856 and 600 texture images, consisting of 640 and 864 texture images	Fusion of color histogram and LBP-based features	TBIR	Image	0.841
Our Proposed Approach	MS COCO	26000 images with 80 classes	Object Detection using Darknet-53 and YOLOv4	CBIR	Text	0.74

Table 5.9: Comparison of various implementations of Swift (here, ✓=Yes and X=No)

Methods	DevStack	Traditional Swift	SwiftStack	Proposed
Metadata	✓	✓	✓	✓
Image Content Extraction	X	X	X	✓
Elasticsearch	✓	X	✓	✓
Search Optimization	X	X	✓	✓
Search based on Contents	X	X	X	✓

5.6 Discussion and Comparative Analysis

The model we propose represents a novel approach to image searching in OpenStack Swift. As a result, conducting a direct comparison with other Swift models becomes challenging. Our model adopts a middleware approach to enhance search efficiency and promote object-awareness throughout the storage system. This middleware component improves the speed and effectiveness of searching within the object storage, simplifying the process of locating and retrieving specific objects. Additionally, it enhances the system's object awareness, offering increased functionality and flexibility in managing and accessing objects. Overall, our proposed solution brings significant improvements to OpenStack Swift, enhancing the effectiveness and functionality of the object storage system.

However, for the purpose of comparison, we divide the comparison into two sub-sections. Firstly, we compare how different Swift models perform searching based on specific parameters. Secondly, we compare the performance of content-based image search models to our proposed model.

5.6.1 Different Swift Models

Table 5.9 presents a comparison of different implementations of Swift using various techniques for searching, along with the parameters set for effective image search in Swift storage. This comparison allows for an evaluation of the strengths and weaknesses of each implementation, providing insights into how they perform in terms of image search functionality.

In Table 5.7, we compare the user availability level of the different implementations based on the model's availability and scalability to work in different environments.

Table 5.10: Resolution check for different uploaded images showing no change in the image quality (SSIM 100% and VQM 100%) after the images are passed through the detection algorithm

Number of Object	Image	Resolution Before Uploading	Resolution After Uploading	SSIM	VQMT
Single	horse.jpg	640 x 425	640 x 425	1	1
	flower.jpg	501 x 640	501 x 640	1	1
	aeroplane.jpg	640 x 425	640 x 425	1	1
Multiple	truck_with_car.jpg	640 x 425	640 x 425	1	1
	cat_on_laptop.jpg	640 x 480	640 x 480	1	1
	man_on_horse.jpg	640 x 426	640 x 426	1	1

5.6.2 Different CBIR Engines

Table 5.8 provides a comparison between our proposed model and different models from related works, focusing on the features extracted from images and the search methods employed. Additionally, the precision of these various models is compared to assess their performance.

Regarding the image upload process, it is important to note that when an image is uploaded to the server and processed by YOLOv4 or YOLOv8, the image itself does not undergo any loss or degradation. In Table 5.10, we present the resolutions of the images, along with the SSIM (Structural Similarity Index) and VQMT (Video Quality Measurement Tool) results before and after the image upload process.

5.7 Conclusion and Future Work

In our work, we combine machine learning features with OpenStack Swift to develop an enhanced solution for efficient searching. By leveraging Elasticsearch, we successfully integrate all components of the design. While our primary objective is to address the searching method in Swift, we also introduce a secondary objective of creating a user-centered content-based image searching system [27]. This system utilizes a text-based database where users have the flexibility to manipulate the YOLOv4 and YOLOv8 algorithms according to their preferences. Importantly, this secondary objective does not compromise the performance of the Swift storage or the Elasticsearch cluster, as they operate independently of each other.

By incorporating the YOLOv4 and YOLOv8 algorithms, which support object detection for both images and live video feeds, we offer users a wide range of choices to cater to their specific needs and requirements. This inclusion adds versatility and expands the functionality of our system, catering to different types of users.

In this chapter, we address the limited focus on content-level metadata search techniques in OpenStack Swift literature. To overcome this limitation, we externally integrate an object detection framework and an Elasticsearch cluster with our Swift storage. Through a series of tests, we assess the viability and responsiveness of our model. While the results demonstrate minimal delay, we acknowledge that further improvements can be made.

As a result, we have identified below future goals to enhance the robustness and user-friendliness of our system:

- Desktop-based Application: We plan to integrate the whole system more compactly using a desktop-based application.
- Authentication Token System: We aim to add an authentication token system in our Elasticsearch server to keep the documents safe from unauthorized access.
- Storing Live Video Feeds: Our target is to use our system to store live video feeds in order to find out the viability of our system as a state-of-the-art video surveillance application.
- Optimization for Faster Performance: We aim to optimize the system to achieve even faster performance in terms of searching and retrieving objects. This may involve refining algorithms, fine-tuning parameters, and exploring parallel processing techniques to reduce response time.
- Enhanced User Interface: We strive to improve the user interface of our system to make it more intuitive and user-friendly. This includes enhancing the search functionalities, providing clear feedback and suggestions, and optimizing the user experience for smooth navigation and interaction.
- Scalability and Flexibility: We plan to enhance the scalability and flexibility of our system to accommodate larger datasets and diverse user requirements. This involves designing efficient data storage and retrieval mechanisms, incorporating advanced indexing techniques, and enabling seamless integration with other tools and platforms.

By focusing on these future goals, we aim to make our system more robust, efficient, and user-friendly, thereby enhancing the overall searching experience within OpenStack Swift.

Part III: Storage Sustainability through Middleware Placement and Orphan Garbage Data Deletion

Chapter 6

Object Storage Sustainability through Removing Offline-Processed Orphan Garbage Data

6.1 Introduction

With myriad diversified applications, multimedia communication over cloud is gaining a great interest in recent times [1, 202, 203]. Such communication entails general user services as well as special user services such as services to management personnel. The management personnel can be law enforcing agency people, crowd monitoring authority persons, etc. A classical example in this regard is the authority of Hajj crowd monitoring authority [30]. A use case for the authority for our focused context is shown in Figure 6.1. To serve such use cases, promising multimedia based cloud systems are now emerging [11, 12]. These systems often leverage various open-source Object Storage Systems (OSS) for faster and easier access to image and video type data, which are the two foremost ingredients in multimedia communication over the Internet.

Here comes the necessity of Object Storage Sustainability in a long run with respect to the continuous growth of unstructured data. Besides, Object storage Systems data management and communication is highly dependent on middlewares design and placement of the middeware in proxy or storage servers [3]. For ensuring data availability, multiple copies of big data is stored in OSS. Hence, regular syncing

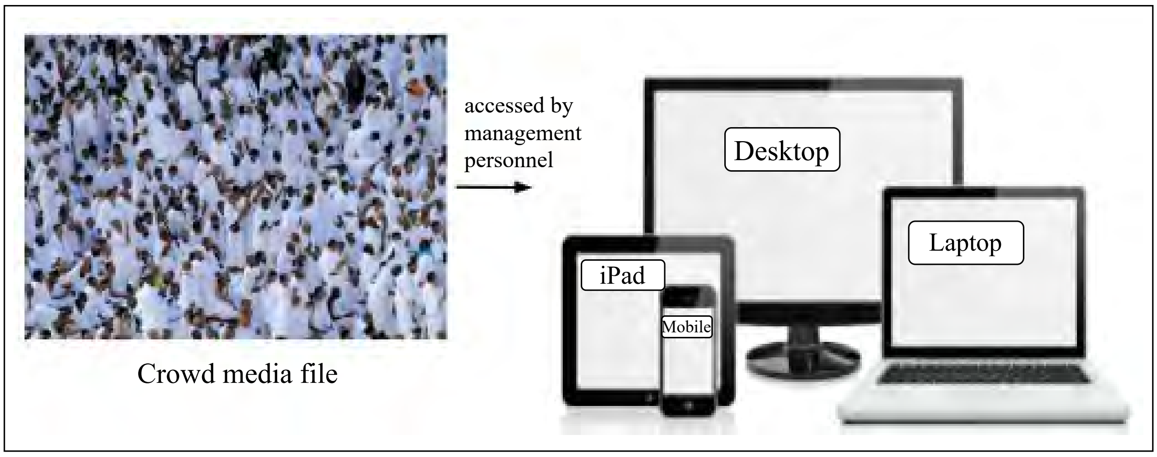


Figure 6.1: A use case of offline processing media data storage. Here, several crowd media files are accessed from cloud storage by the Hajj management personnel using several diversified remote devices whenever needed. Hence, different versions of media files (images and videos) are stored in the cloud storage using offline processing beforehand.

and checking is necessary for finding bit rot and file degradation to ensure long-term preservation storage. Several studies focus on data storage sustainability and their impact to ensure long-term sustainability to avoid undesirable consequences [28, 29].

Besides, several applications such as crowd management, real-time location-aware services, and medical systems need to access multimedia data from diversified remote devices (in Figure 6.1). As an example, crowd management of millions of pilgrims for performing Hajj, Umrah, and Kumbh Mela is challenging and appropriate processing and communication from the cloud is a must [30, 31]. Hence, context-aware and location-aware cloud-based frameworks and services are emerging [204]. These frameworks need both online and offline processing of unstructured data such as images and videos. Additionally, real-time video streaming is another prominent feature for managing these kinds of services using cloud infrastructures [205].

Similarly, video experiences slower responses from important sites, as the sizes of video files are generally much higher than that of corresponding image files. Many video streaming service providers in this regard provide their services using cloud-based video storage systems. Here, efficient cloud-side operation management is needed for ensuring different features such as smooth video streaming, dynamic adaptive streaming, etc. Besides, proper and updated video segments need to be supplied from the cloud storage systems to achieve the features. In this regard, Recent studies focus on several methods of mobile and web streaming [202, 203, 206, 207], gateway-based shaping methods for HTTP adaptive streaming (HAS) [208], quality of experience of HAS [209], optimal transcoding and caching for adaptive streaming in content delivery networks [210], etc. However, none of these studies focuses on video streaming support for a cloud specifically for an Object Storage System, which is now treated as a well-adopted solution for cloud service development.

Yet another aspect worth investigating for image and video delivering clouds is efficient usage of the cloud storage. Due to diversification in operations and usages, there can arise different types of data and objects in such a storage. For example, components such as client database, AUTH server database, etc., can produce data and objects that will be never required at all. To be more specific, in the multimedia cloud storage [11, 211], there can be different versions of data for images and videos that are never going to be accessed by a user.

Irrespective of the future requirements, all the data or objects of a storage are generally considered to be an asset for the cloud storage, as data storage has no concern about what data is stored or whether the data is necessary or not. However, there are some other components such as client database, AUTH server database, etc., which are always necessary components for designing a full system. Therefore, in reality, all data in the cloud storage may not always be an asset for other components. For example, a user can upload some personal images to a media cloud system. All the versions of the images were uploaded successfully, however, when returning the response the network got disconnected (in Figure 6.2).

Hence, there will be no information about these images in the AUTH server where all the lists of files are stored for users. This data can be useful for cloud storage, however, never be used for users' purposes. We use a new term for this kind of data - *orphan garbage data* - to imply such data to be garbage as well as having no effective linkage to its ancestor. Such data gets generated by different types of cloud operations, e.g., when a cloud operation produces different versions of the same data. This happens in the case of offline processing media clouds that produce different versions of data both for images and videos. Such productions result in orphan garbage data in a multimedia cloud storage.

Furthermore, research studies focus on several aspects of such redundant data deletion architecture. Some studies present the memory garbage collector algorithms in big data context [212, 213]. Other studies, Linux container based deletion [36], Smartbin based deletion in wireless sensor networks [37], orphan process detection [214, 215], and assured deletion [38, 39] present some deletion approaches, which are not applicable to the case of orphan garbage data in cloud due to architectural as well as operational mismatches between cloud and the cases focused on these studies.

In summary, research studies are yet to focus on these important realms in multimedia cloud operation and communication covering impact of middleware placement through designing adaptive segmented video streaming, and orphan garbage data deletion and management for ensuring Object Storage sustainability. Therefore, in this Chapter, we first propose a new middleware named 'VideoSegmenter', which is used for making video segments according to any kind of time range using FFmpeg [216]. One specialty of our middleware architecture is that it can give the user/streaming server any playable segment on the fly. Another specialty is the ability to deploy this middleware in the object server rather than in the proxy server in OpenStack Swift.

Besides, we propose a novel approach *'RemOrphan'* for detecting and deleting orphan garbage data in a multimedia cloud. Here, we develop a deletion daemon to find and remove orphan data in an efficient manner to make the data storage sustainable along with enhancing CPU usage. In proposing all these new techniques, we present a video segmenter middleware, impact of middleware placement, and a deletion daemon to eventually perform the tasks from the same OpenStack-like system. We evaluate performance of the system in a real setup comprising a server in Canada and a client in Bangladesh. Our rigorous experimental results demonstrate that we can achieve up to 30% lower video segment download time, 30% reduced network overhead, and 25% reduced sync delay through utilizing our proposed techniques.

Based on our study, we make the following set of contributions in this Chapter:

- We analyze two important factors related to long-term Object Storage sustainability - impact of middleware placement and orphan garbage data in Object Storage System.
- We design a new middleware 'VideoSegmenter' for supporting HTTP adaptive streaming in OpenStack Swift-like systems such as SPMS. Accordingly, we implement a new package using setup-tools [217], which can be easily integrated in the existing OpenStack Swift. We analyze and present that the proposed middleware should be deployed in the object server for getting faster responses and for avoiding extra overhead and maintaining long-term sustainability.
- We present a new technique for finding out unused orphan garbage data in multimedia storage

systems, which poses a great threat for sustainable storage systems. This orphan data is responsible for unnecessary sync delay in replication and extra network overhead related to replica (r) and number of objects per node (n).

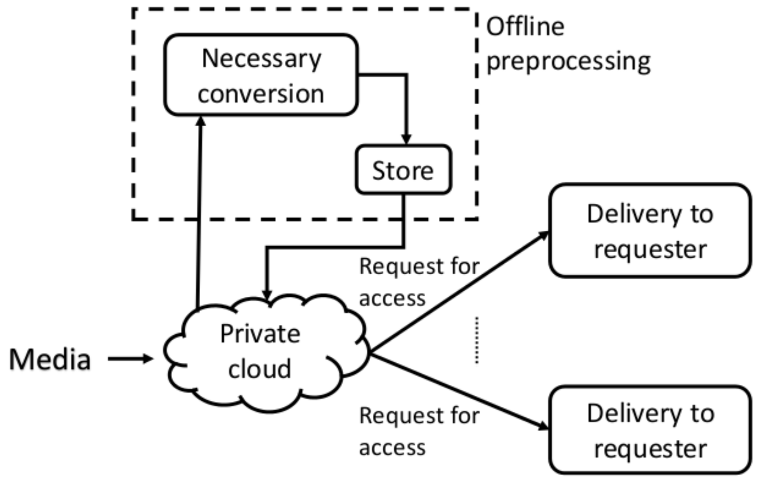
- Moreover, we design a deletion daemon named *'RemOrphan'* for removing the orphan data using OpenStack rings and scripts in an efficient way. Our custom deletion daemon presents a configurable solution that offers options to run it once or in a periodic manner.
- Finally, we perform rigorous experimental evaluation of our proposed techniques in a real testbed comprising a high-configuration server in Canada and a client in Bangladesh. Our experimental results confirm efficacies of all our proposed techniques against that of classical alternatives.

The organization of this Chapter is further segmented into different sections. Section 6.2 contains the literature reviews of recent papers related to our study. After, in Section 6.3, we present three important concepts - an OpenStack Swift like object storage system, middleware in OSS, and the definition of orphan garbage data. Next, system design and implementation is presented in Section 6.4. Besides, Section 6.5 contains experimental test-bed setup and performance evaluation. Furthermore, in Section 6.6, we present the discussion and comparative analysis of our proposed methodology with the existing literature. Finally, the conclusion and future prospects of this research are stated in Section 6.7 respectively.

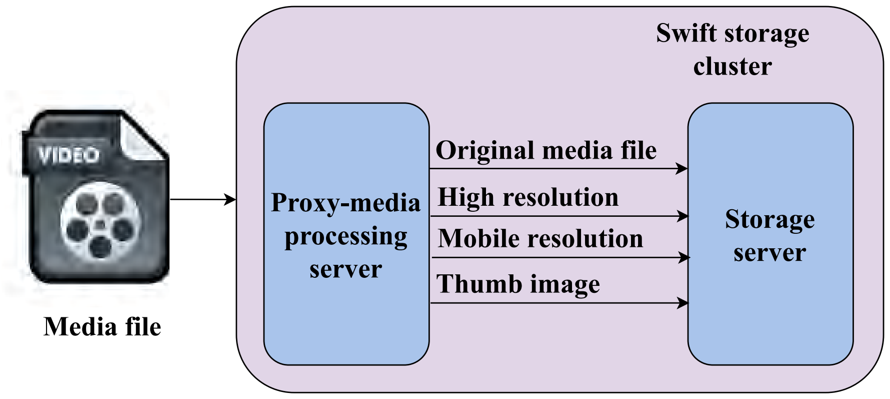
6.2 Related Work

In this Section, we present the related studies exploring object storage sustainability, on-demand video segmentation and streaming and orphan garbage data deletion. In the literature, we find very few studies on object storage sustainability. Study [28] demonstrates that long-term preservation storage requires more than just storing multiple copies of a file. It is also essential to regularly check those copies for bit rot and other types of degradation. Therefore, file integrity tools are necessary to ensure the ongoing integrity of the stored data. Another study [29] presents that the utilization of big data at a massive scale is likely to result in some negative repercussions. While some of these consequences can be predicted, others may be completely unforeseeable. This essay focuses on the sustainability-related issues that arise from the implementation of big data.

Besides, efficient and smoother video streaming, dynamic adaptive streaming, etc., are some special



(a) Offline processing in media cloud [11]



(b) Media file is converted to different resolutions to support diversified devices

Figure 6.2: Offline processing models for storing and retrieving media files from the cloud. Here, multiple versions of media files are processed and stored in the cloud. During the processing time, orphan data may be stored in the cloud.

features, which need both cloud server side and streaming server side operation management. For achieving these features, studies [11, 12, 53] mainly focus on faster and secure management of media files through designing middlewares. Due to a demand hike of video streaming services over the mobile networks, the wireless link capacity fails to cope up with the growing traffic load resulting in poor service quality of video streaming. A research study in this regard [207] constructs a private agent for each active mobile user in the cloud to adaptively adjust the video quality utilizing scalable video coding technique based on the feedback of link condition and social network interactions.

Furthermore, study [203] has developed a new framework called EMS for streaming ultra-high-definition video. This framework combines erasure-coded storage with multi-source streaming. Moreover, they created two metrics, one for deadline awareness and the other for latency sensitivity, to measure the quality of service provided by video servers. Additionally, they propose a federated learning approach to adaptively update the service quality, which includes a reinforcement learning based multi-server selection process for local user training, and a global aggregation of service quality. Study [202] introduces a new approach called Segment Prefetching and Caching at the Edge for Adaptive Video Streaming (SPACE). They have developed and analyzed several segment prefetching policies that vary in terms of resource usage, required player and radio metrics, and deployment complexity.

Moreover, study [218] presents the WVSNP-DASH framework, which relies on video segments that can be played independently and have a particular naming syntax that conveys elementary metadata. This system facilitates flexible search, transfer, distribution, and playback of the video segments. To enhance the adaptive video streaming performance in CCN, study [219] suggests a hop-by-hop adaptive video streaming approach known as HAVS-CCN. Other studies focus on several methods of mobile and web streaming [206], gateway-based shaping methods for HTTP adaptive streaming (HAS) [208], survey on quality of experience of HAS [209], etc.

In addition, existing CDNs may not be sufficiently cost effective for distributing adaptive video streaming due to the lack of orchestration on storage, computing and bandwidth resources. Hence, a research study [210] leverages the notions of media cloud to deliver on demand adaptive video streaming services, where those resources can be dynamically scheduled minimizing the total operational cost by optimally orchestrating multiple resources. For this, study formulates and utilizes an optimization problem by examining a three-way trade-off between the caching, transcoding, and bandwidth costs. However, there is no study on video streaming support for object storage systems and the impact of middleware placement for storage sustainability. Furthermore, the research study [38] presents a notion of SmartBin in place of old-fashioned practice such as hiring people to regularly check and empty filled dustbins. SmartBin, integrates the idea of IOT with Wireless Sensor Networks. Another study [39] discusses the need for assured deletion in cloud along with identifying cloud features that pose a threat to assured deletion and describes various assured deletion challenges as well. Besides, study [220] focuses on analyzing the GREEDY Garbage Collector strategy under the condition of uniformly independently distributed write accesses.

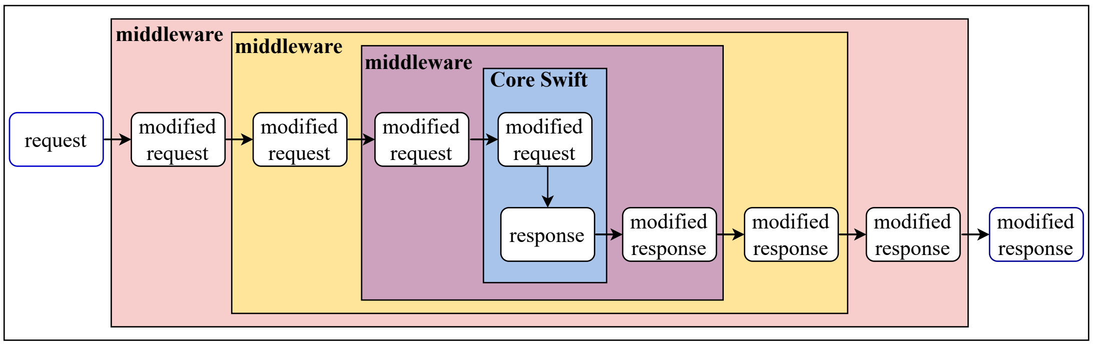


Figure 6.3: How different layers of middleware work in Web Server Gateway Interface (WSGI) for Object Storage System

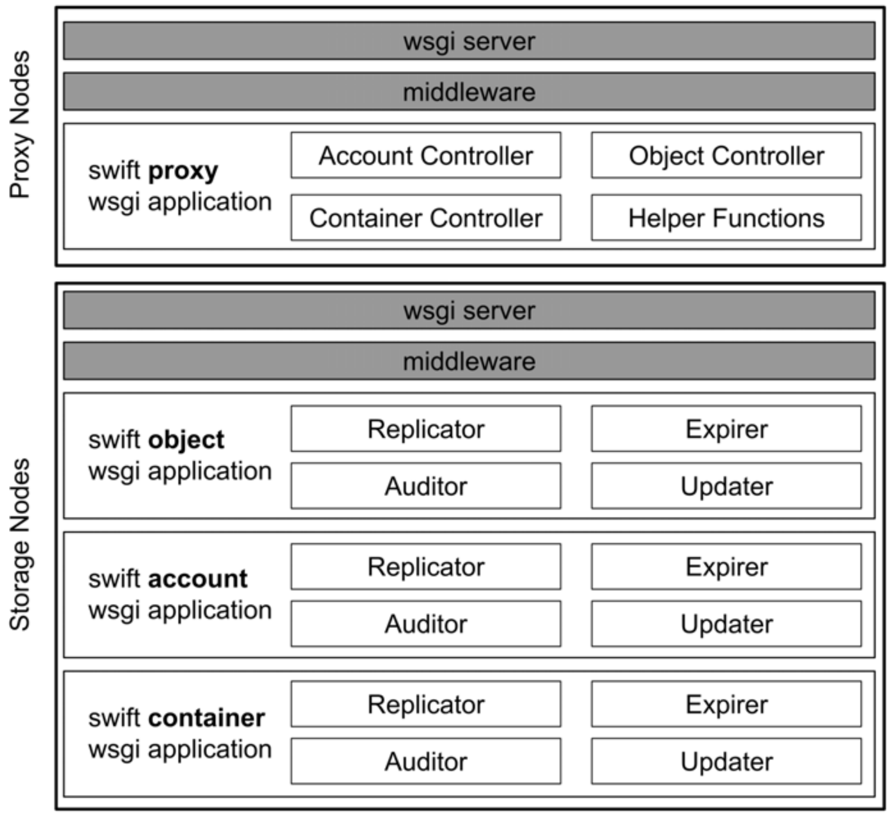


Figure 6.4: Different consistency processes and layers in proxy and storage nodes of OpenStack Swift

Moreover, study [213] examines existing Big Data platforms and their memory profiles to investigate why traditional algorithms, which remain widely used, are inadequate. It also evaluates newly suggested memory management algorithms that are specifically designed for Big Data environments [221–225]. The research assesses the scalability of these recent memory management algorithms by comparing their throughput (improvement in application throughput) and pause time (reduction in application latency) to that of classic algorithms. Besides, study [212] performs a thorough evaluation of three widely used garbage collectors, namely Parallel, CMS, and G1, using four typical Spark applications. Their evaluation involves a comprehensive analysis of the relationship between the memory usage patterns of these big data applications and the GC patterns of the collectors, leading to several insights into GC inefficiencies. Based on the findings, the authors provide empirical guidelines for application developers and offer useful optimization strategies for developing garbage collectors that are suitable for big data environments.

On the other hand, in distributed systems, orphan processes may be generated as a result of remote procedure calls (RPC). There are two types of orphans: crash-orphans, which occur when the client crashes, and abort-orphans, which occur when the parent process is aborted. Orphan processes are problematic because they consume system resources and can result in inconsistent data. To address this issue, several studies develop new methods for detecting orphan processes [214, 215]. However, none of these studies deal with the problem of removing orphan garbage data.

To the best of our knowledge, our proposed methodology is the first to focus on video streaming data retrieval and impact of middleware placement in object storage systems. Besides, we present an architecture named *'RemOrphan'* for orphan garbage data detection and deletion to maintain a healthy and sustainable storage, which is yet to be focused in the literature.

6.3 Background

Our goal is to focus on object storage sustainability through analyzing the impact of middleware placement and orphan garbage data deletion. Hence, in this section, at first, we present a media cloud storage system named SPMS which is designed using OpenStack Swift. After, we describe the details on middlewares in OSS. Finally, the definition of orphan garbage data is presented using appropriate examples.

6.3.1 SPMS (Secure Processing-aware Media Storage)

Recently, many media cloud storages have been deployed using OpenStack Swift. Swift is an open-source object storage system having some special features such as eventual consistency, high availability, fault tolerance, replication, etc. Swift has two types of servers - proxy for management and

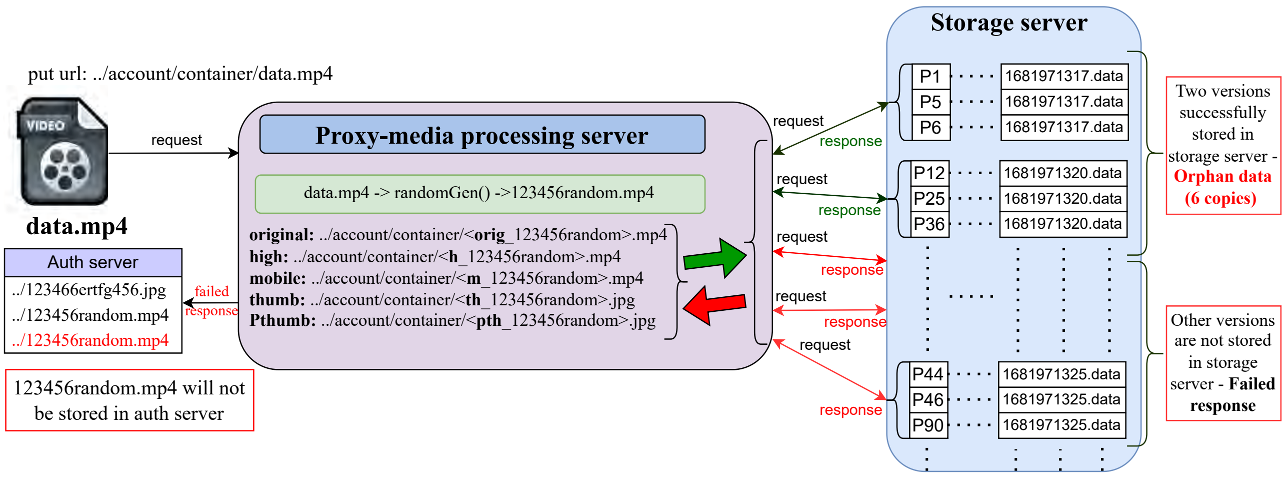

put url: ../account/container/data.mp4
VIDEO
data.mp4
request
Proxy-media processing server
data.mp4 -> randomGen() ->123456random.mp4
original: ../account/container/<orig_123456random>.mp4
high: ../account/container/<h_123456random>.mp4
mobile: ../account/container/<m_123456random>.mp4
thumb: ../account/container/<th_123456random>.jpg
Pthumb: ../account/container/<pth_123456random>.jpg
Auth server
../123466ertfg456.jpg
../123456random.mp4
../123456random.mp4
failed response
123456random.mp4 will not be stored in auth server
request
response
Storage server
P1 1681971317.data
P5 1681971317.data
P6 1681971317.data
P12 1681971320.data
P25 1681971320.data
P36 1681971320.data
P44 1681971325.data
P46 1681971325.data
P90 1681971325.data
Two versions successfully stored in storage server - Orphan data (6 copies)
Other versions are not stored in storage server - Failed response

Figure 6.5: How orphan garbage data are created due to network disconnection, client timeout problem, object versioning, etc. Here, data.mp4 file is uploaded from the client. For this single video file, five versions are uploaded in the storage nodes having three copies for each versions. Two versions are successfully stored while background processing is done, however, other versions are not uploaded successfully due to different reasons. Hence, the final response is failed and the url is not stored in AUTH database. The above six copies are orphan garbage data, which are still in the storage server without any use whatsoever.

processing, and storage servers (account, container, and object) for storing database and data objects [3]. SPMS, which is designed using OpenStack Swift through adding several new middlewares. SPMS system has every feature of Swift. Additionally, it also has some special features of media securing, image data conversion to PJPEG, image resizing to multiple dimensions, and video transcoding and resizing to various sizes [11]. As SPMS-like media storage systems are used for multipurpose media management tasks such as video streaming and storing of many versions of objects, the tasks of optimizing multimedia retrieval and orphan garbage data deletion comes into play to ensure long-term sustainability.

6.3.2 Middleware in Object Storage System

An Object Storage System like OpenStack Swift is built on Python's Web Services Gateway Interface (WSGI) model and configured using the Python Paste framework. In the WSGI model, middlewares are a vital part and they are designed to pass the requests through several layers to reach the core application. Besides, middleware wraps other middlewares one by one down to the core application in the center without knowing anything about the other layers. Hence, developing middleware codes are easy and straightforward for the developer to design new features.

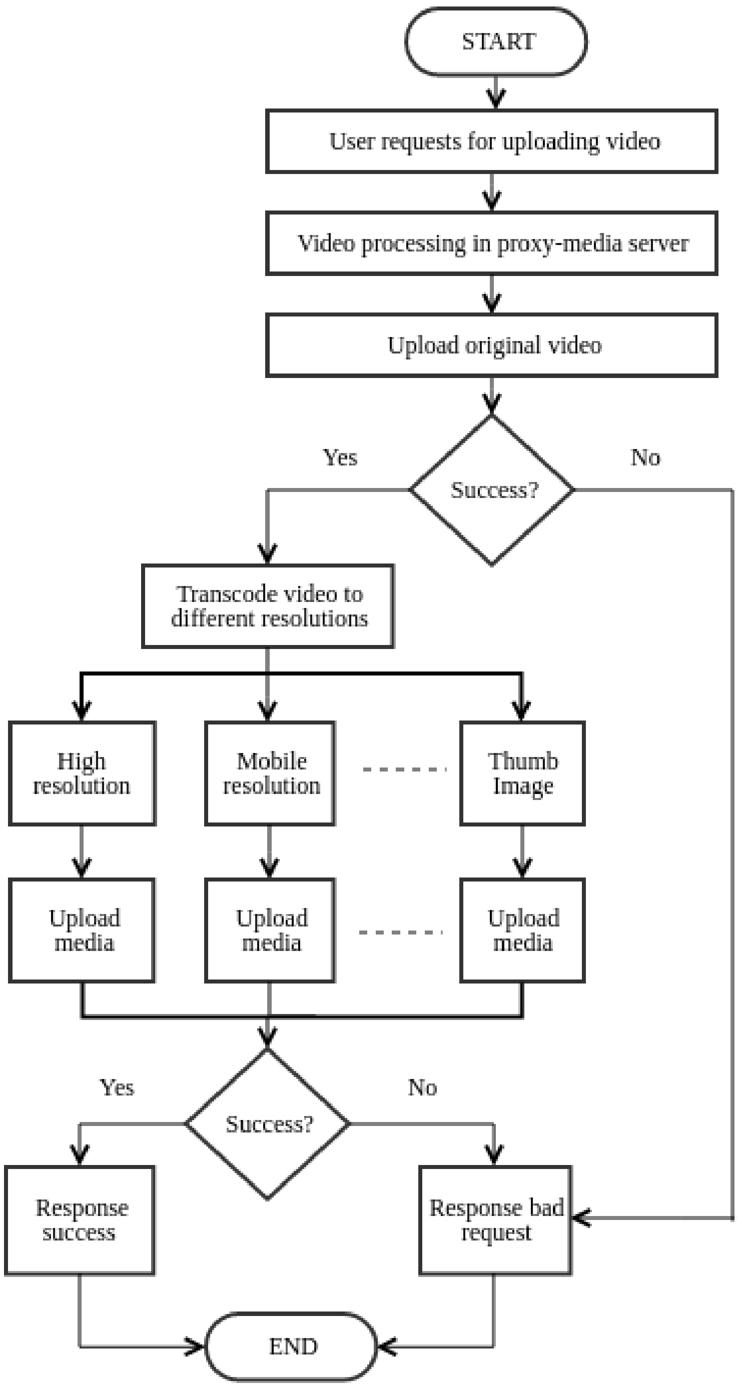


Figure 6.6: Case-1: ‘Response after all uploading’

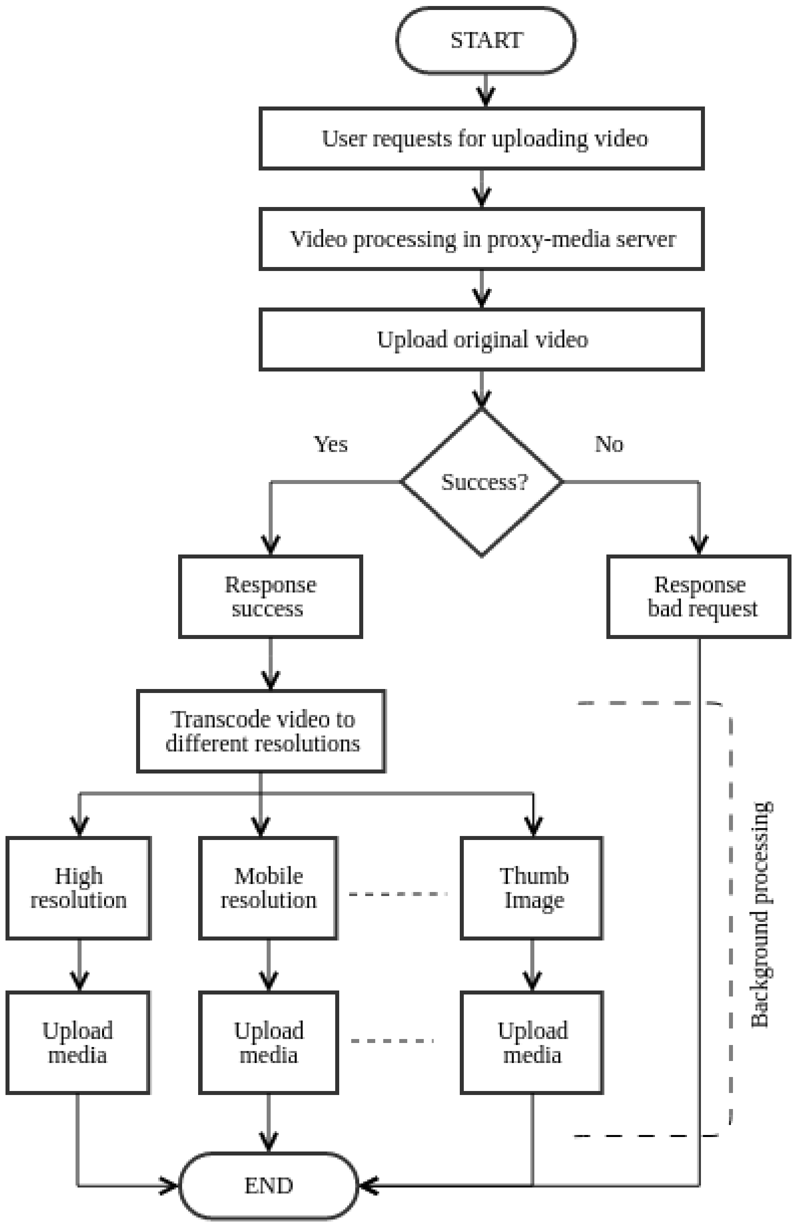


Figure 6.7: Case-2: 'Quick response with background processing'

Figure 6.3 presents how a middleware system with multiple layers works. When a user request enters the system, the request is potentially altered by each middleware layer as it moves inward towards the final processing by core application. The response then travels back out the layers of middleware, with each layer having the option to modify the response. Each middleware layer can either modify a request or let it pass through unchanged. Finally, the final response is returned to the user. Each middleware layer can inspect, modify, or short-circuit a request or response. Different features are implemented using middleware. In OpenStack Swift, processing related tasks are handled in the proxy server. There are other consistency services to maintain replication, audit, update of objects in both proxy and storage servers. Figure 6.4 presents different consistency processes and layers in proxy and storage nodes (servers) of OpenStack Swift. Here, Each node consists of middleware layers to perform several tasks. Hence, middleware placement is a concerning issue for long-term sustainability and efficiency of an Object Storage System.

6.3.3 Orphan Garbage Data

Data management with optimization of CPU and memory usage in data centers is now the most challenging topic for data scientists. Any kind of data is valuable, as all the data in a storage system can be used for further processing or mining purposes. A big question is now, is there any garbage data in the cloud? For example, resizing of images and transcoding of videos according to several resolutions for covering diversified remote devices and data versioning must be needed in the media cloud. To do so, recent studies present offline processing models for storing and retrieving media files from the cloud (in Figure 6.2a). Therefore, for a single object, there are several different resolutions of the original one (in Figure 6.2b). The data which is never used for quite a long time, is referred to 'unused data'. This data can be archived using erasure coding policy or different kinds of mechanisms. However, there is some kind of data which exists in cloud storage, which has no information both on the client side or in the AUTH database. This can happen for network disconnection, client timeout problems, etc. This is the orphan garbage data which can be a great threat for data storage sustainability.

Moreover, several studies introduce and design middleware for image and video processing tasks in the Object Storage System. In SPMS-like systems, for covering diversified remote devices, many versions of images (200, 300, 600, etc., to aspect ratio) along with progressive JPEG images are stored. Same applies for video files, e.g., high-resolution (720 to aspect ratio) video files, mobile-resolution (400 to aspect ratio) video files, etc. Besides, study [211] presents that the different versions of photos viewed in Facebook is around 80-100 Billion. Among them, the Thumbnails version is viewed around 10.2%, the Small version is 84.4%, 0.2% is the Medium version, and the Large version is 5.2%. This is one of the main reasons for getting orphan garbage data created, where no information either in client side or in AUTH database about the orphan data gets stored. The non-existence of information can occur due to network disconnection, client timeout problem, object versioning, etc., [11, 53]. Furthermore, transcoding large video files into different resolutions is a challenging task, hence, some research studies propose background processing rather than waiting for all the versions to upload and send a successful response.

In study [11], the researchers propose two different designs to convert and upload high resolution, mobile resolution, and other versions along with the original video file. In the first case (Case-1: 'Response after all uploading'), the system sends response after successfully uploading all the versions, hence it takes much longer time and the possibility of failed response is higher (in Figure 6.6). On the other hand, in the second case (Case-2: 'Quick response with background processing'), the system sends a success response immediately after getting a success response for the original video file, and starts background processing for all other versions (in Figure 6.7). For both cases, the system may create orphan garbage data due to network disruption, client-timeout, internal server communication error, and so on.

We present an example of how orphan garbage data is created due to network disconnection, client timeout problem, object versioning, etc. Here, data.mp4 file is uploaded from the client (in Figure 6.5). Then, for maintaining security and integrity, the file name is changed using some random function. After, for this single video file, five requests are sent from proxy server to storage/object server having three copies for each version. Two versions are successfully stored when background processing, but somehow other versions are not uploaded successfully due to several reasons. In the storage node, the object is stored using the system's own convention (i.e ¡epoch-time¿.data). Hence, the final response is failed and the URL is not stored in the AUTH database. However, the six copies which are successfully stored through internal communication from proxy to storage nodes, will remain as the orphan garbage data.

6.4 System Design and Implementation

We propose a general architecture for managing multimedia data smoothly and efficiently in media cloud storage systems. Hence, first, we describe the background of our proposal. Then, we present

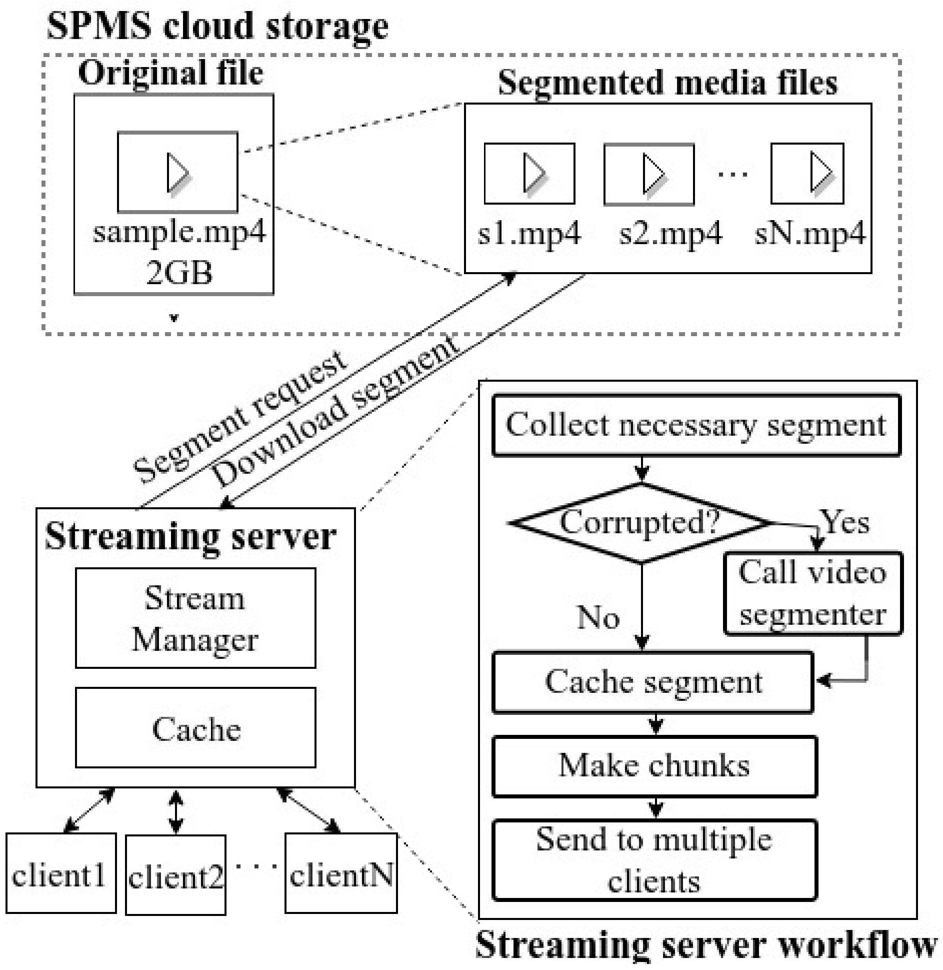


Figure 6.8: Our proposed architecture of VideoSegmenter middleware, which presents how a streaming server requests for a segment from the cloud storage

our proposed architecture in the following subsections.

6.4.1 Video Segmenter Middleware

Nowadays, smoother and efficient video streaming including dynamic adaptation of streaming has become popular for its diversified usages. To understand its underlying methodology, first, we need to know how streaming works in a large system. Figure 6.8 presents the architecture of how a streaming server and cloud system gets interconnected while streaming videos to multiple clients. Here, first the streaming server collects necessary segments or full files from the storage server. Then, the streaming manager creates small chunks from the segment for sending those chunks to the clients. There is another component named viewer server which is responsible for publishing the chunks to clients.

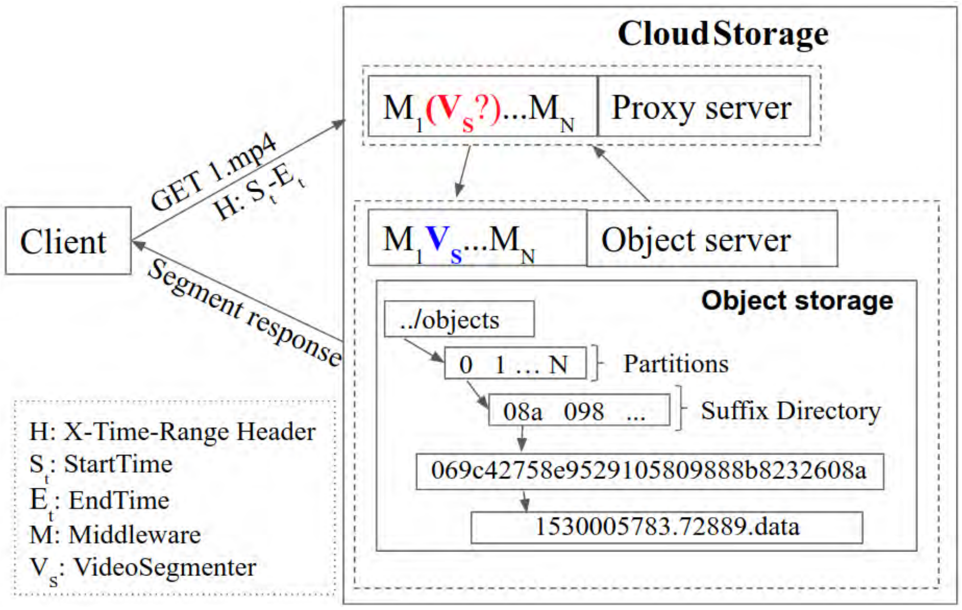


Figure 6.9: An example of segment GET request using proposed X-Time-Range-Header (VideoSegmenter middleware (V_s) is deployed in the object server)

It is worth mentioning that, for streaming a video, the streaming server needs all information of the video so that it can download the full video or any segments from the original file. For saving unnecessary processing, the storage server emphasizes on storing segments of the original file beforehand. In this case, when the storage server has no segment stored or the downloaded segment is found to be corrupted, 'VideoSegmenter' middleware comes into play. We design the middleware having it similar to that for partial range requests. The only difference is that our design takes input of a time range header, X-Time-Range: startTime-endTime. Such a range request is supported by HTTP protocols and it only gives some bytes within the requested range. However, segment requests of our proposed middleware provide any playable segment within the requested time range. Figure 6.9 presents a time range request example of our proposed middleware along with object storage architecture of OpenStack Swift.

As Swift has its own architecture, predefined variables, iterators, etc., for downloading an object, we need to investigate Swift object storage implementation very deeply and find out which parts of the implementation need to be refactored when returning segments rather than the full object. Moreover, to enable ease of deployment and maintenance, our target is not to change the open source code, rather to deploy a new middleware package so that it can be deployed and integrated easily with the

Algorithm 5 Algorithm for the VideoSegmenter middleware

```
1: procedure VideoSegmenter($app, env, start\_response$)    ▷ Each middleware has a start_response method which need status and headers of the response
2:     $timeRange \leftarrow env.get(TimeRangeHeader)$
3:     if $requestMethod = GET$ and $timeRange$ then
4:         $itr \leftarrow app(env, start\_response)$    ▷ Here, itr is a dictionary having important attributes in OpenStack Swift, or list insatance if any error occurs
5:         if $isinstance(itr, list)$ then
6:             return itr
7:         end if    ▷ isinstance, list both are python keyword
8:         $sT, eT \leftarrow timeRange.split()$
9:         $validTimePattern \leftarrow CheckRegex$    ▷ Check valid time pattern using Regular expressions
10:        $outF \leftarrow SegmentedFileLocation$
11:        $outputFp, etag \leftarrow SegmentVideo(itr.\_data\_file, outF, sT, eT)$
12:        $itr.\_fp \leftarrow open(outputFp, rb)$
13:        $itr.\_diskfile.\_data\_file \leftarrow outputFp$
14:        $itr.\_obj\_size \leftarrow os.path.getsize(output\_fp)$
15:        $itr.\_etag \leftarrow etag$
16:        $UpdateHeader()$    ▷ According to new content-length and new etag of video segment
17:        $start\_response(staus[0], headers[0])$
18:        return itr
19:    end if
20:    return app
21: end procedure
```

system.

Accordingly, we find that two classes (BaseDiskFile and DiskFileReader) from two separate files in OpenStack Swift (/usr/lib/python3.8/site-packages/swift/obj/mem_diskfile.py, /usr/lib/python3.8/site-packages/swift/obj/diskfile.py) are responsible for downloading an object. We change the values according to new segment size, etag (MD5 hash in this files of the downloading object), and location ($\_fp$, $\_diskfile.\_data\_file$, $\_obj\_size$, $\_etag$). We present the VideoSegmenter algorithm here (Algorithm 5 and 6).

Subsequently, we consider another key aspect- as the proxy server is responsible for all the processing, can we deploy VideoSegmenter in the proxy server? Here, the problem is, for segmenting a video file, we need the full file first. Hence, the proxy server needs to download the full file in some temporary location and then clip the video to send segments to the requester. This procedure is slower and needs more network overhead. On the contrary, if we deploy VideoSegmenter in the object server, full object downloading is not needed any more as the clipping is done directly on data location. Here we

Algorithm 6 Algorithm for segmenting video file

```
1: procedure SegmentVideo($inP, otP, sT, eT$)  ▷ Here, i = Input file path, o = Output file path, sT = startTime, eT = endTime
2:     $temp, fileWOExt \leftarrow inP.rsplit()$
3:     $swiftExt \leftarrow .data$
4:     $videoExt \leftarrow .mp4$
5:     $otP+ = join(fileWOExt.clip(swiftExt), sT, eT)$
6:     $matchedFile \leftarrow glob.glob(otP)$
7:     $tStamp \leftarrow time.time()$
8:     if ($mathedFile$) then
9:         $pathWBF, etag \leftarrow matchedFile.rsplit()$
10:        $otP \leftarrow join(pathWBF, etag, tStamp) + videoExt$
11:        $os.rename(matchedFile[0], otP)$
12:        return otP, etag
13:    else
14:        $tempPath \leftarrow otp + randomInt + videoExt$
15:        $clipCmnd \leftarrow FFmpegCommand$  ▷ FFmpeg command for clipping the video segment using sT and Et [216]
16:        $output, error \leftarrow subprocess.Popen(clipCmnd).communicate()$
17:        if $output$ then
18:            raise Exception
19:        end if
20:        $etag \leftarrow md5(tempPath)$  ▷ New etag calculation for segment video
21:        $otP \leftarrow join(otP, str(etag), str(tStamp)) + videoExt\ os.rename(tempPath, otP)$
22:        return otP, etag
23:    end if
24: end procedure
```

need r times processing in all replicated locations of the original object. However, the segment will automatically be deleted from its location after a predefined time in this case.

All the middleware are written in /usr/lib/python3.8/site-packages/swift/common/middleware/ package of OpenStack Swift. If we add a new middleware there, then upgradation of a new release will be required making the implementation process complex. Hence, we implement a new distribution of VideoSemter middleware using python setup-tools [217]. Besides, in our deployed SPMS object server, we change the $object - server.conf$ file and add VideoSegmenter egg file for including our new middleware ($use = egg : video\_segmenter\#video\_segmenter$ in paste.filter_factory [226]). In the next subsection, we delineate orphan data deletion daemon pertinent to both video and image data.

6.4.2 '*RemOrphan*': Orphan Data Deletion

The data that is never used for quite a long time is referred to as unused data. This data can be archived using some erasure policy or using different types of mechanisms. However, there is another kind of data called orphan data or garbage data, which exposes a great threat for data storage sustainability. Main reasons for the threat of orphan data includes storing of each object using some random names, existence of different types of object versions [11, 53], network connection timeout with client or AUTH database, etc. Hence, we propose an architecture for detecting such orphan data using OpenStack Swift hash rings and scripts.

It is worth mentioning that the main two design goals of OpenStack Swift and similar systems are eventual consistency and high availability through replicated data objects across multiple nodes. A research study in [227] presents how sync delay and network overhead get related when $r > 3$ and $n >> 1000$. Hence, a high number of objects are always responsible for sync delay and network overhead. Moreover, if this scenario happens for some unnecessary orphan data, then it appears to be a great loss for storage service providers.

To tackle this problem, we propose some key steps in our orphan data detection and deletion daemon. 1) We collect all data list from client/AUTH database for certain time interval. 2) Then, we collect all object lists of all accounts from the Swift account and container database for the same time interval. 3) We create black list and white list from all versions of the objects. 4) We delete black listed files using bulk DELETE request [3]. Figure 6.10 presents the flow diagram of orphan data deletion daemon.

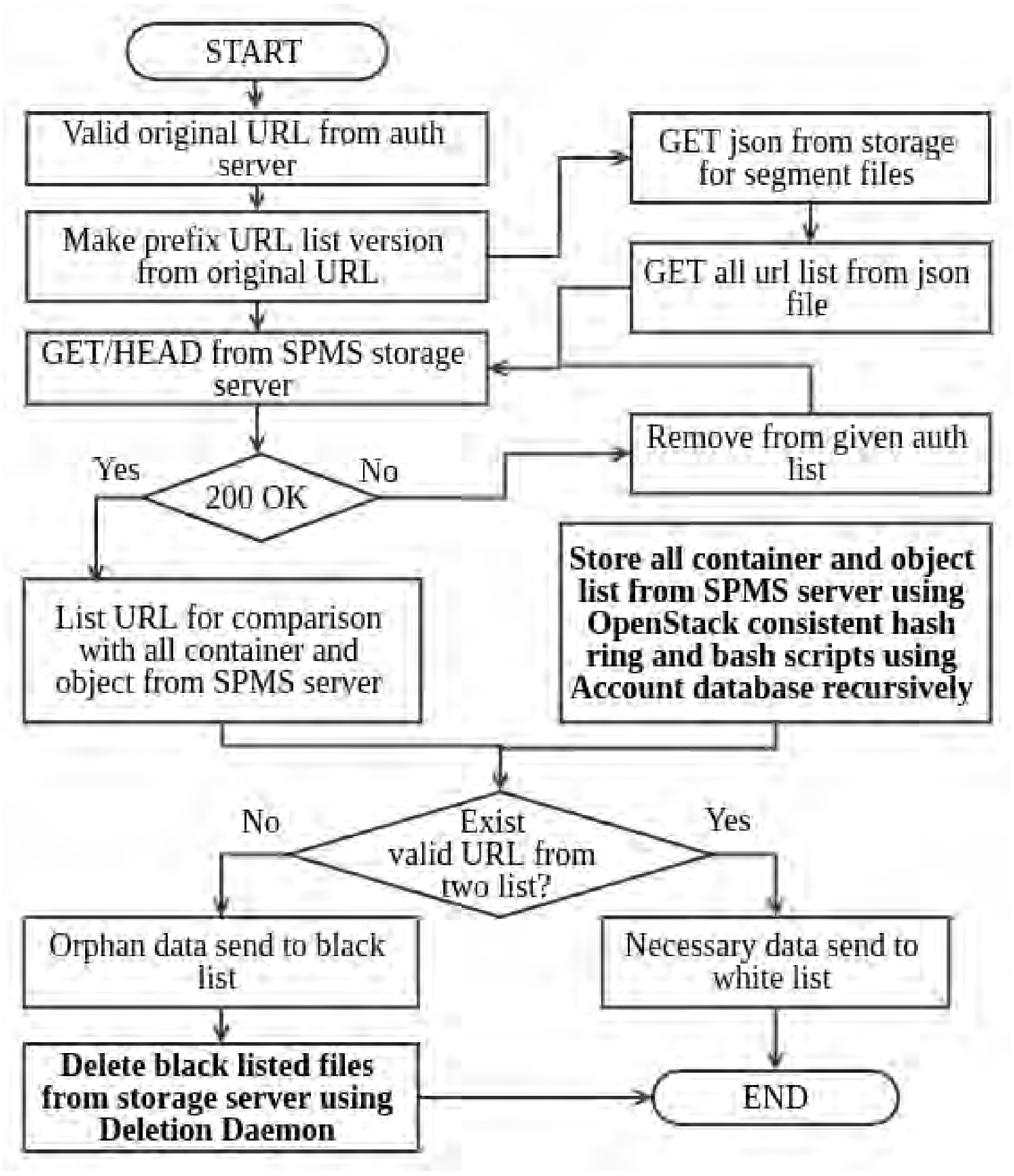

START
Valid original URL from auth server
Make prefix URL list version from original URL
GET json from storage for segment files
GET all url list from json file
GET/HEAD from SPMS storage server
Yes
200 OK
No
Remove from given auth list
List URL for comparison with all container and object from SPMS server
Store all container and object list from SPMS server using OpenStack consistent hash ring and bash scripts using Account database recursively
No
Exist valid URL from two list?
Yes
Orphan data send to black list
Necessary data send to white list
Delete black listed files from storage server using Deletion Daemon
END

Figure 6.10: Flow diagram for orphan data deletion daemon

Our custom daemon server is configurable from the perspective of its considered time interval for collection of lists. This offers us options to run once or in forever mode by daily, weekly, or monthly based on our configured time interval.

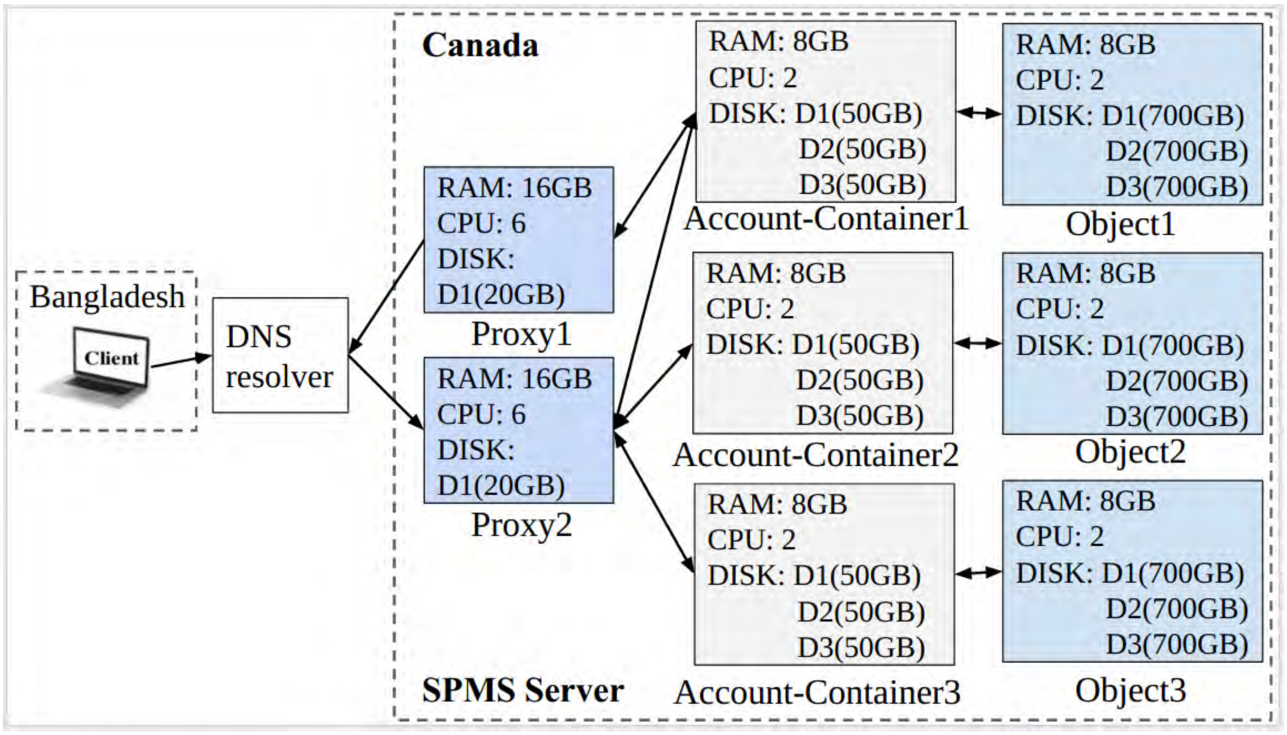

Canada
Bangladesh
Client
DNS resolver
RAM: 16GB CPU: 6 DISK: D1(20GB)
Proxy1
RAM: 16GB CPU: 6 DISK: D1(20GB)
Proxy2
RAM: 8GB CPU: 2 DISK: D1(50GB) D2(50GB) D3(50GB)
Account-Container1
Account-Container2
Account-Container3
RAM: 8GB CPU: 2 DISK: D1(700GB) D2(700GB) D3(700GB)
Object1
Object2
Object3
SPMS Server

Figure 6.11: Test bed setup servers in Canada and client in Bangladesh

6.5 Performance Evaluation

We evaluate performance of our proposed architectures through a real implementation. We, first, briefly present our experimental testbed setup. Then, we present our experimental results and findings for our three different architectures.

6.5.1 Experimental Testbed Setup

We use real high-resource machines for deploying testbed servers in Canada. Here, we use two proxy servers, three account-container servers, and three object servers for the media storage cluster having model AMD Opteron 62xx class CPU, and OS CentOS 7. The memory and disk configurations of our Swift servers here cover- 1) two proxys each having one 8 GB memory and one 20 GB disk, 2) three account-containers each having one 8 GB memory and three disks each of 50 GB, and 3) three objects having one 8 GB memory and three disks each of 700 GB. Each server has six 1 GB network interface cards. Figure 6.11 and Table 6.1 present the experimental setup of our testbed.

In addition, we deploy a private media cloud SPMS using OpenStack Swift (stable newton branch)

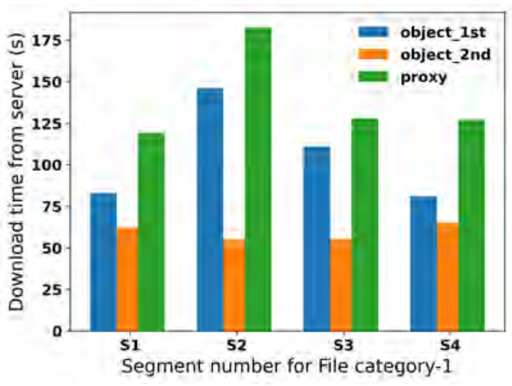
Table 6.1: Configuration of machines used in testbed setup

Informations	Proxy server	Object server	Account-container server	Client machine
Architecture	x86_64	x86_64	x86_64	x86_64
CPU(s)	16	48	16	1
On-line CPU(s) list	0-15	0-47	0-15	0
Thread(s) per core	2	1	2	1
Core(s) per socket	4	12	4	1
Socket(s)	2	4	2	1
NUMA node(s)	2	8	2	1
CPU family	6	16	6	6
Model name	Intel(R) Xeon(R) CPU E5620 @2.40GHz	AMD Opteron(tm) Processor 6174	Intel(R) Xeon(R) CPU E5620 @2.40GHz	QEMU Virtual CPU version 1.5.3
CPU MHz	2394.141	2199.967	2394.103	2393.998
Virtualization type	VT-x	AMD-V	VT-x	full Storage

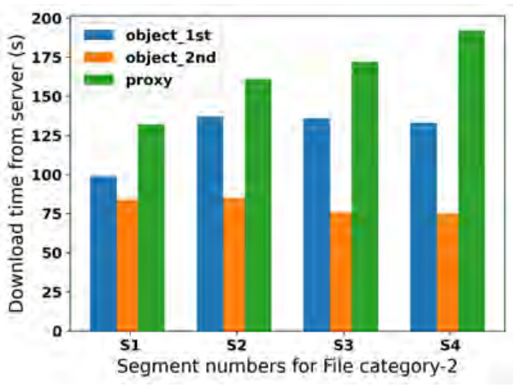
with three replicas ($r = 3$) and 16384 partitions ($p = 16384$). There are nine devices for account, container, and object ring file, hence, each device has around 5461 partitions in $/srv/node/ < server >$ folders (devices are mount in this location according to OpenStack Swift guide [3]). Here, a server can be an account, container, or object. Moreover, we implement a social site for both mobile and web users, and then upload different types of data from clients to the development server for around eight months. Besides, in order to evaluate our proposed architecture, we upload large media files from a benchmark video surveillance data set. We use 125 videos ranging in size from 3.8 MB to 1.4 GB and upload the videos in a periodic manner. Furthermore, we create 10,000 accounts and 10,000 containers in Swift cluster and upload around 1M images and video files in those accounts. Hence, no of objects (n) are 1M for our testbed server. We upload around 1.5TB data, and hence, total data becomes $1.5TB \times 3$ in our development server.

Table 6.2: Demographic information of three file categories

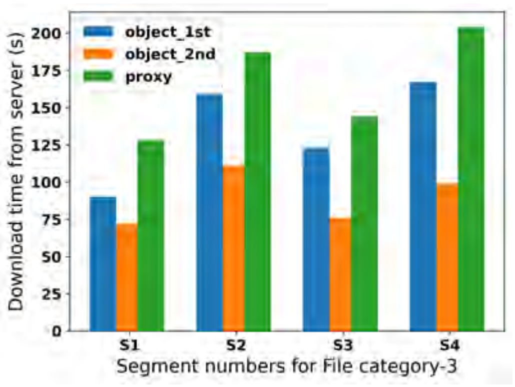
File category	Avg. Size (GB)	Avg. Duration (min)
Short file	0.49	94.85
Medium file	0.66	152.2
Long file	0.87	177.7



(a) Download time for four segments for file category-1



(b) Download time for four segments for file category-2



(c) Download time for four segments for file category-3

Figure 6.12: Comparison of download time for segments of three different file categories. S1, S2, S3, and S4 denote the average segment of 10 and 15 minutes of 1st to 4th segments respectively. In the graph, 1st bar (blue) represents the download time of the segment from the object server at first time request, 2nd bar (orange) represents the same segment download time from object server at second time request. Besides, 3rd bar (green) represents the download time of the same segment from the proxy server.

6.5.2 Experimental Results

For testing VideoSegmenter middleware, we make different segments of 10 minutes and 15 minutes of different video files (Category-1 of having the average size 0.49GB and average duration 94.85

Table 6.3: Time improvement percentage status for different segments with respect to retrieving the segment from 2nd time versus 1st time from the object server, if we place the middleware in the object server. Moreover, segment download time comparison is presented by placing the middleware in the proxy server (object vs proxy).

File category	**% Improvement (1st segment)**	
	Object_2nd vs Object_1st	Object vs Proxy
Short file	25.3	30.25
Medium file	15.15	25
Long file	20	29.69
	% Improvement (2nd segment)	
	Object_2nd vs Object_1st	Object vs Proxy
Short file	62.33	20
Medium file	37.96	14.91
Long file	30.19	14.97
	% Improvement (3rd segment)	
	Object_2nd vs Object_1st	Object vs Proxy
Short file	50.45	13.28
Medium file	44.12	20.93
Long file	38.21	14.58
	% Improvement (4th segment)	
	Object_2nd vs Object_1st	Object vs Proxy
Short file	19.75	36.22
Medium file	43.61	30.73
Long file	40.72	18.14
Avg time improvement	**Object_2nd vs Object_1st: 22.39%**	**Object vs Proxy: 35.65%**

minutes; Category-2: average size 0.66GB and average duration 152.2 minutes; Category-3: average size 0.87GB and average duration 177.7 minutes). We deploy VideoSegmenter middleware in both proxy server and object server. There was some difference in the middleware code for the proxy server. In our proposed architecture of VideoSegmenter for object server, we store or cache the segment for around one to two days in object segment location. Hence, a download request of the same segment (second download request onward) needs lower time than the first time request from the object server. Figure 6.12 and Table 6.3 present a comparison of download times from object server and proxy server for three different categories. Here, we show download times for different segments of different files. We take 100 iterations for each single segment for each type of download, and present an average of the 100 iterations in the graph. The download times correspond to three different types of download- 1) download from the object server for the first time, 2) download from the object server for the second time, and 3) download from the proxy server. As Figure 6.12 demonstrates, downloading from

Table 6.4: Orphan garbage data deletion status per node for testbed server

1st round - Deletion status

Metrics	Before deletion	After deletion	Improvement
Object count	1M	650K	lower 35% data
Sync delay	1440s	1080s	lower 25% delay
Network delay	500MB	350MB	lower 30% overhead

2nd round - Deletion status after one year

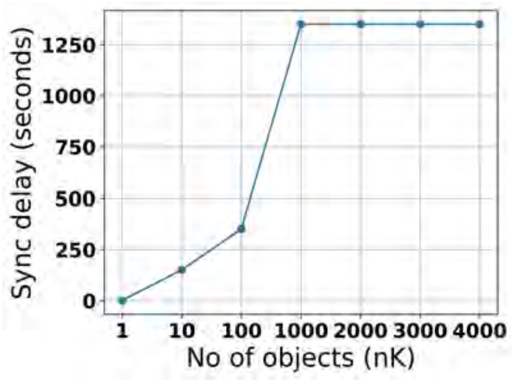
Metrics	Before deletion	After deletion	Improvement
Object count	1G	630M	lower 30% data
Sync delay	1280s	965s	lower 25% delay
Network delay	500MB	350MB	lower 30% overhead

3rd round - Deletion status after two days

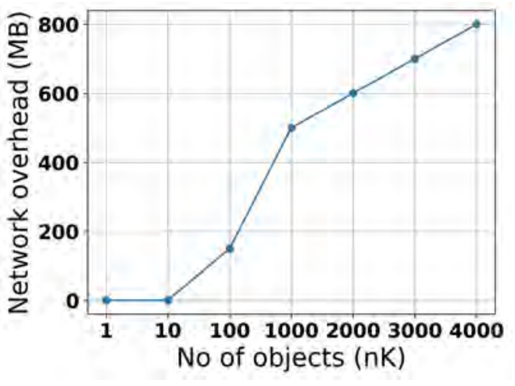
Metrics	Before deletion	After deletion	Improvement
Object count	700M	678M	lower 4% data
Sync delay	1280s	1100s	lower 20% delay
Network delay	500MB	350MB	lower 30% overhead

the object server always takes much lower time than that from the proxy server. Nevertheless, the second download from the object server takes lower time than the first download utilizing the caching.

Furthermore, Table 6.4 presents that around 35% data is orphan data according to our setup testbed at the first round when we run the deletion daemon. After removing the orphan data, sync delay and network overhead get lower by up to 25% and 30% respectively. Next, several users upload a bulk amount of images and videos regularly and we run the deletion daemon after one year again. Table 6.4 presents the values after removing the new orphan data in the second experiment. Furthermore, we run the deletion daemon after two days of the second experiment. Table 6.4 presents the values after running the deletion daemon a third time.



(a)



(b)

Figure 6.13: Relation between sync delay and network overhead with respect to the number of objects per node (n). Here, the value mentioned as nK (in x-axis), i.e., 10 values represent 10,000 objects.

Table 6.5: CPU and memory overhead for video segmentation and orphan data deletion

Metrics	Video Segmentation	Orphan data deletion
CPU	3%	14%
Memory	5%	16%

6.5.3 Overhead Analysis

Figure 6.13 presents the relation between sync delay in seconds and network overhead in MB with respect to the number of objects (n) per node/server when replica $r = 3$. When n grows, then the sync delay increases with r. Here, an interesting observation is that, when $n > 1M$, the sync delay increases quite slowly (for a fixed r). This happens owing to the number of partitions [227]. When $n > 1M >> 2.5M$ (for a fixed r), though the number of sync messages keeps stable, the size of each sync message still grows with n and each sync message contains more hash values of more data objects. Hence, network overhead continues to grow with n when $n > 1M$. Besides, Table 6.5 presents the memory and CPU overhead of overall architecture.

6.6 Discussion and Comparative Analysis

This study, for the first time in the literature, establishes the necessity of object storage sustainability as long-term storage has diverse effects such as performance, efficiency, energy consumption, fault

Table 6.6: A qualitative comparison of related research studies according to several features such as algorithm type, memory and storage management, developers deployment effort, orphan data collection, middleware placement and deployment, etc. (here, GC refers to Garbage Collection)

Research study	Algorithm	Management	Deployment effort	Platform	Focus metrices	Middleware placement	Orphan garbage collector
Althaus et al., 2022 [220]	Greedy Garbage Collection	Memory GC	Medium	Solid State Drives (SSDs)	Throughput	X	X
Noor et al., 2022 [1]	Modified scan scripts and DCT quantization	Storage management, no orphan data deletion	Low	Object storage	Throughput, Latency	✓	X
PokeMem (Kweun et al., 2022) [228]	Modified GC	Memory GC	Medium	Processing (Enhanced Spark)	Throughput	X	X
Noor et al., 2021 [12]	Resizing and security enforcement	Storage management, no orphan data deletion	Low	Object storage	Throughput, Latency, Concurrency	✓	X
GC-CR (Louati et al., 2017) [36]	Checkpoint-Restart	Decentralized GC (snapshot)	High	Storage	Latency	X	X
iCSI (Kim et al., 2017) [229]	Lightweight VM collector	Cloud Garbage VM Collector	High	Stoarge	Latency	X	X
NG2C (Bruno et al., 2017) [230]	Modified GC	Memory GC	Low	Processing, storage	Latency	X	X
Deca (Lu et al., 2016) [222]	Unmodified GC	Memory GC	High	Processing (Spark)	Throughput	X	X
Taurus (Maas et al., 2016) [221]	Unmodified GC	Memory GC	Low	Processing, storage	Latency	X	X
Broom (Gog et al., 2015) [224]	Modified GC	Memory GC	High	Processing (Naiad)	Throughput	X	X
FACADE (Nguyen et al., 2015) [225]	Unmodified GC	Memory GC	Low	Iterative processing	Throughput	X	X
NumaGiC (Gidra et al., 2015) [223]	Modified GC	Memory GC	None	Processing, storage	Throughput	X	X

Research study	Algorithm	Management	Deploy-ment effort	Platform	Focus metrices	Middle-ware place-ment	Orphan garbage collec-tor
DSA (Cohen and Petrank et al., 2015) [231]	Modified GC	Memory GC	Medium	Processing, storage	Throughput	X	X
Sabbaghi et al., 2013 [215]	Orphan process detection	Remote Procedure Call	None	Processing	Fault tolerance	X	X
C4 (Tene et al., 2011) [232]	Modified GC	Memory GC	None	Processing, storage	Latency	X	X
Jahanshahi et al., 2005 [214]	Orphan process detection	Remote Procedure Call	None	Processing	Fault tolerance	X	X
G1 (Detlefs et al., 2004) [233]	Modified GC	Memory GC	None	Processing, storage	Latency	X	X
***RemOrphan* (our proposed)**	**Orphan garbage data collector**	**Storage manage-ment**	**Low**	**Proce-ssing, object storage**	**Through-put, Latency, Concur-rency, Fault tolerance**	✓	✓

tolerance, and so on. In this Section, we discuss some important aspects of our study and present comparison of related research studies according to several features (in Table 6.6). We illustrate the features such as algorithm type, memory and storage management, developers deployment effort, intended platform, focus metrics, orphan data collection, middleware placement and deployment, etc. In the algorithm feature, we discuss what kind of algorithm the studies present in their work. Some studies use Modified Garbage Collector algorithm, i.e., they extended the traditional Garbage Collection algorithms [223, 224, 228, 230–233]. On the other hand, several studies do not modify the traditional Garbage collection algorithms, hence we refer to them as Unmodified Garbage Collector [221, 222, 225].

As the management perspective, we discover mainly four types of categories - Memory Garbage Collector [220–225,228,230–233], Garbage VM Collector [36,229], Orphan Process Collector [214,215], and Orphan Garbage Data Collector. However, no existing literature focuses on the later issue i.e. orphan garbage data. Besides, we present the system deployment effort as high, low, medium, and none with respect to the algorithms proposed in recent studies. Next, the target platform and performance metrics are presented in the comparison table. Most of the studies focus on the metrics - throughput and latency. However, study [12] works on throughput, latency, and concurrency whereas our proposed architecture improves the throughput, latency, concurrency and makes the system more fault-tolerant. By adopting a completely new methodology, we can achieve long-term object storage sustainability through analyzing the proper middleware placement and removing orphan garbage data regularly using deletion daemon. The time needs for listing and copying a directory using OpenStack Swift hash rings is $O(mlogN)$ and $O(n + logN)$ respectively. Hence, the time complexity of our proposed *'RemOrphan'* algorithm is $O(mlogN) + O(n + logN)$. Here, N is the total number of files in the file system, n represents the number of files stored in a certain directory, and m is the number of direct children under a certain directory.

6.7 Conclusion and Future Work

In this Chapter, we delineate three important realms related to multimedia data management on the cloud and the diverse effect on storage sustainability. Here, we point three key vacancies in the literature comprising retrieval of video streaming data, middleware placement based on their responsibility, and detection and deletion of orphan garbage data (a new type of data that is of no use however retained for a long time over cloud storage).

Hence, we design a new middleware in object server for downloading a time interval playable video segments which can be easily integrated in OpenStack Swift and similar systems such as SPMS. Furthermore, we propose a mechanism for removing orphan garbage data from cloud storage. We perform rigorous experimentation over a real setup established in Canada and accessed from Bangladesh. Our experimentation covers both system level and subjective evaluations. The evaluation results confirm that we can achieve substantial performance improvement using our proposed mechanisms.

Our future work includes exploration of SSYNC for account, container, and object servers using multiple replicas. We also plan to explore recursive deletion daemon algorithms using different hash rings. Besides, experimenting over different server setups with large scale simulations is yet another aspect worth investigating in future.

Part IV: System-level Load Testing of Cloud Storage Ecosystem

Chapter 7

svLoad: An Automated Case-Driven Load Testing in Cloud Systems

7.1 Introduction

In this era of connected devices, the demand of storage systems are increasing exponentially [3]. Using various open-source projects [3, 234], several distributed private cloud storage systems are now emerging [11, 53]. At the same time, clients are increasingly demanding faster and easier access to data from these systems. In addition, system designers need to test the behavior of these distributed architectures under massive operational loads to designing architectures properly and flawlessly. Furthermore, service providers use cache(s) in front of backend servers for retrieving data faster from distributed private cloud systems. Hence, information about the time elapsed for retrieving data from cache or backend in different test scenarios is necessary to design a reliable system. Analyzing that information, service providers can find out numbers and appropriate locations of the cache and backend servers for achieving the best outcome. Apart from these, load testing is also needed to tune the parameters of the software, hardware, and network used in the system.

As of today, private and public cloud service providers design their own distributed storage systems using several data centers. Choosing best locations for deploying cache and backend cloud servers in data centers is one of the challenging task for most service providers. Here, time delays in object uploading and downloading are directly related to how the cache and backend servers are distributed.

Furthermore, for successful deployment of distributed architectures including caches and clouds in production environments, proper load testing is mandatory. As such, several existing research studies focus on load testing tools and architectures based on performance and functional testing criteria. The study in [40] proposes an empirical testing by monitoring user experience and system health in a feedback loop between traffic shifts. Other studies [41, 235, 236] propose automated approaches to validate whether a performance test resembles the field workload or not. Unfortunately, these studies propose and analyze only real-time test cases without focusing on network and software tuning using the outcomes of the test analysis.

Furthermore, recent studies do not focus on finding a general load testing architecture for testing distributed systems that combine both cache and backend servers. Hence, in this Chapter, by means of a rigorous study we propose a load test architecture 'svLoad' that facilitates the process of finding out the best values of parameters based on real scenarios. We also locate the bottlenecks of OpenStack Swift and Varnish cache servers when they are operating with extensive load. In our study, we offer extensive load requests from some predefined clients based on our proposed test cases. Hence, the server will be busy on handling the requests. At the same time, we send concurrent download requests using bash scripts and observe percentage of success rates among the requests, and how much time they need for ending up with successful responses. Besides, we identify resource utilization bottlenecks in both system and network performances, and tune the system and networks parameters accordingly, and analyze the system behavior through subsequent load tests.

Based on our study, we make the following set of specific contributions in this Chapter:

- We propose 20 different test cases based on diversified real scenario covering different protocol types (HTTP or HTTPS), URL types (same or different URLs) , load types (with or without loads), and server types (backend, cache) for performing load test on cloud systems using several tools - JMeter [237], Ansible [238], and our custom bash scripts.
- We perform continuous rigorous load testing on two open source cloud systems namely Varnish [234] and Swift [3], and find out bottlenecks in the system and network that are worthy of tuning.
- Subsequently, we perform parameter tuning as per our findings of load testing. Here, first we come up with a comparison of response times for downloading files from cache and backend

servers for each test case through rigorous experimentation. Then, we improve the response times and success rates of concurrent requests up to 80% and 90% respectively through our tuning in the Swift, Varnish, and network systems.

7.2 Literature Survey

The main goal of load testing is to identify the upper limit of systems in terms of performance of the database, hardware, network, etc. Hence, realistic test case based architectures for distributed cloud storage system is critical. Also, while, functional tests may ensure the general performance of a cloud, load tests ensure system reliablity and fault tolerance at even very demanding load requests. Load tests give developers confidence that the cloud is well sized. Hence, the importance of realistic and generalized load tests for cache and backend, today.

Furthermore, load tests for open source caches like Varnish [234], and cloud systems like OpenStack Swift [3] are needed for tuning system parameters for vendors who merge these two components for building large distributed cloud architectures. Recently, several works have appeared on load test tools, cloud evaluation criteria based on load test, performance testing of web applications, workload optimization, continuous validation, etc. in this realm. Here, we present a short summary of these works to movitate our new architecture 'svLoad' for testing load capabilities in cache and backend cloud servers.

A comparative performance study [42] among different testing tools shows Webload is better in terms of assessing response time and throughput, compared to tools like Neoload, LoadImpact, Loadster and LoadUI. The study in [239] presented important factors in cloud computing performance, and analyzed and evaluated cloud performance in various scenarios based on criteria, characteristics, and simulation. The study in [235] used the Load Runner testing tool to capture end-user business processes and created automated performance testing virtual scripts to organize, manage, and monitor load testing through running virtual users. Another study in [40] analyzed the behavior of individual systems and groups of systems to identify resource utilization bottlenecks to ensure Facebook's allocated capacity for servicing download requests via tuning.

In addition, studies like [240, 241] analyze system performance degradation or problems handling required system throughput. Studies in [242–245] presented late-cycle measurement-based and model-

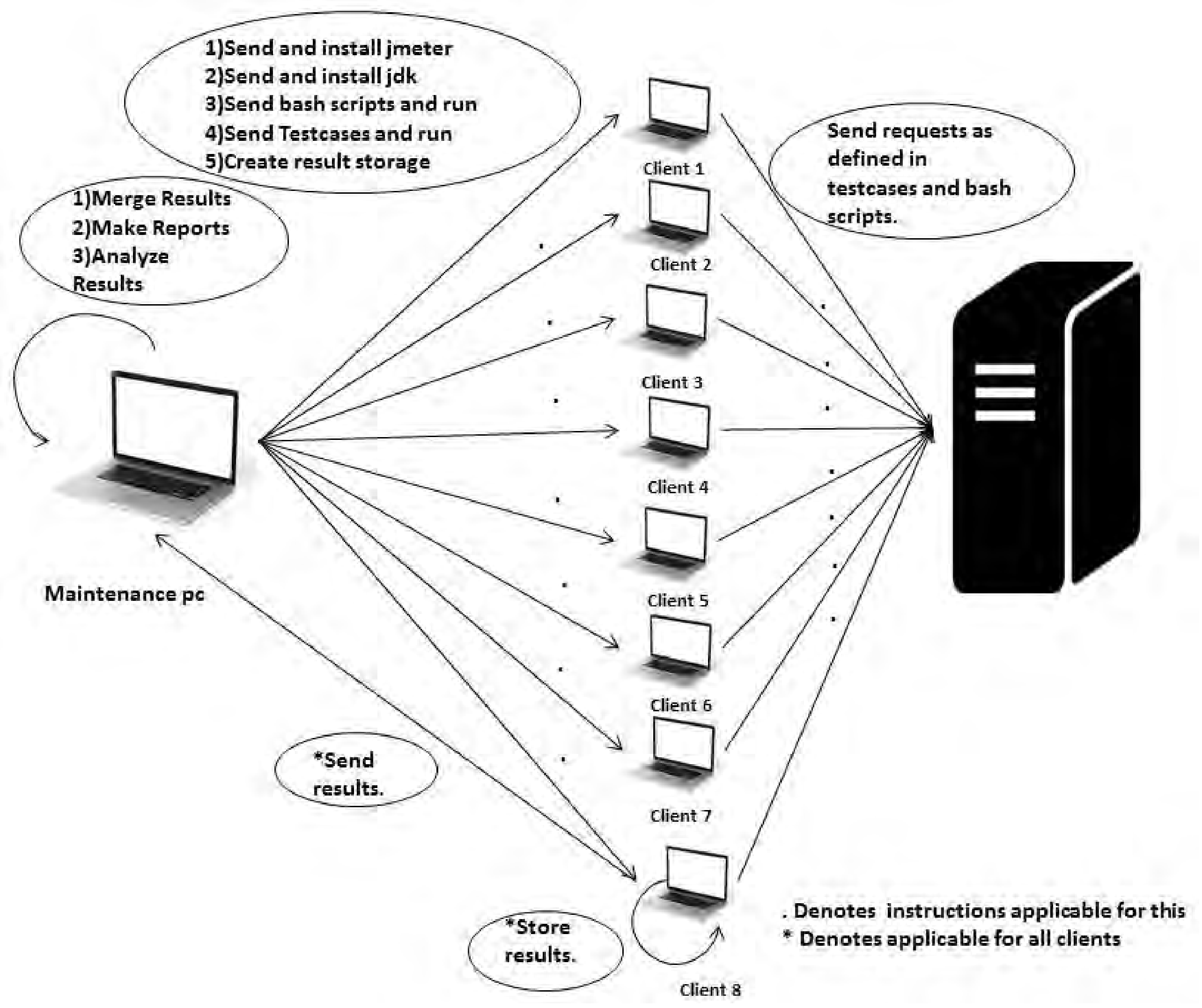


Figure 7.1: Architectural overview of our proposed load test model

based approaches. Measurement-based approaches apply testing, diagnosis and tuning late in the development cycle. The study in [41] presented that performance analysts must continually validate whether their tests are reflective of the field or not. Such validation may be performed by comparing execution logs from the test and the field. After going through all these studies, we can say that these studies on load tests address some real time test cases, and do focus on load test based on automated framework, but they did not cover the general load test scenarios based on real metrics. Furthermore, these studies barely concentrated on network and software tuning along with load tests to make the system best suited while using open source projects such as Swift, Varnish, etc. To the best of our knowledge, load test architectures that vary real metrics such as network protocol, URL type, load

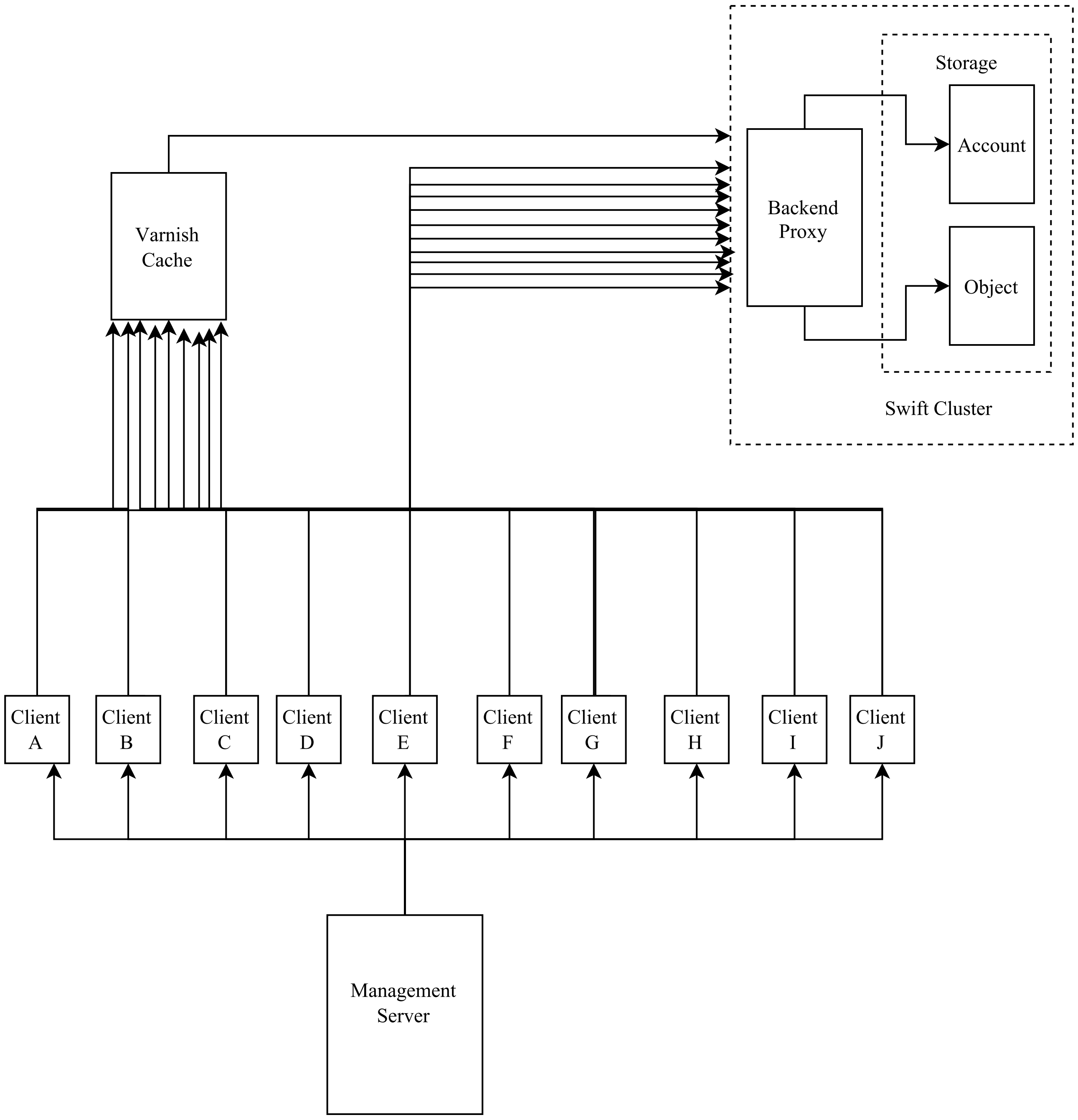


Figure 7.2: Overview of client, backend, varnish, and management servers

amount, and server type using JMeter, Ansible, and use case centric bash scripts as load test aids for load testing in distributed storage system are yet to be accomplished. This motivates our Chapter.

7.3 Proposed Methodology

In this Chapter, we propose a general load test architecture for distributed storage system. Figure 7.1 and Figure 7.2 presents the architectural overview of our proposed load test model. Our proposed operational methodology over this architecture comprises several key steps, which we present in the following subsections.

7.3.1 Load Test Planning

The main technique for measuring performance of servers is to give extensive concurrent requests to respective servers. For this, we need to run the load tests in regular basis as the results vary with software and system variable factors such as network bandwidth, CPU, memory usage, etc. Hence, we focus on automating the whole process with minimal effort. We give importance on several necessary questions as follows:

1) Which tools should be used for load testing purpose?
2) How many machines should be used?
3) What would be the required machine configurations?
4) How to automate the whole process?
5) What would be the most important metrics for designing the test cases?
6) What components of the hardware, software, and network systems would be the limiting factors of the performance?

In recent times, there are several tools for load testing with various use cases. Among them, finding the appropriate tools for serving vendors purposefully is tough. Besides, designing the test bed using available machines, picking the highly configured machines, designing test case scenarios based on real findings are most challenging. Furthermore, for getting better and optimized performance from distributed systems, developers need to find out the hardware, software, and network system components which limit the performance. However, there are many open source software for testing functional behavior and performance. We choose Apache JMeter [237] as a testing tool and Ansible as IT automation engine because as they are simple, powerful and cross platform supportive. We use 10 client machines for giving concurrent loads.

Table 7.1: Test case scenarios for different metrics

Test case ID	Protocol		URL		Load		Server		
	HTTP	HTTPS	Same	Different	Without	With	Backend	Cache	
								Hit	Miss
TC0	✓		✓		✓		✓		
TC1	✓		✓			✓	✓		
TC2	✓		✓		✓			✓	
TC3	✓		✓			✓		✓	
TC4	✓		✓		✓				✓
TC5	✓		✓			✓			✓
TC6	✓			✓	✓		✓		
TC7	✓			✓		✓	✓		
TC8	✓			✓	✓				✓
TC9	✓			✓		✓			✓
TC10	✓			✓	✓			✓	
TC11	✓			✓		✓		✓	
TC12		✓	✓		✓		✓		
TC13		✓	✓			✓	✓		
TC14		✓	✓		✓			✓	
TC15		✓	✓			✓		✓	
TC16		✓	✓		✓				✓
TC17		✓	✓			✓			✓
TC18		✓		✓	✓		✓		
TC19		✓		✓		✓	✓		
TC20		✓		✓	✓				✓
TC21		✓		✓		✓			✓
TC22		✓		✓	✓			✓	
TC23		✓		✓		✓		✓	

7.3.2 Creating Test Scenarios

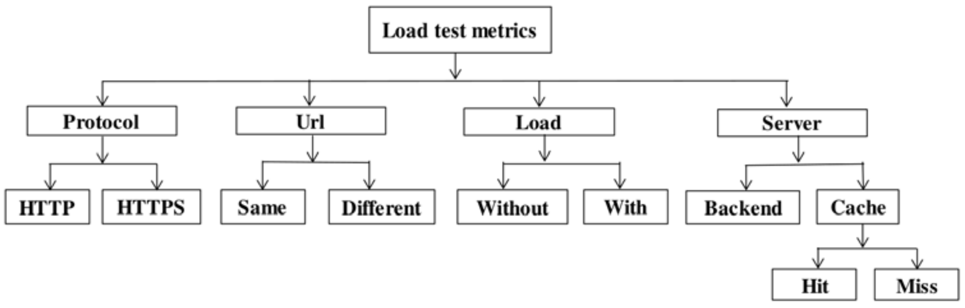


Figure 7.3: Test case hierarchy of proposed load test metrics

Our strategy is to make servers busy with highest concurrent loads. In the meantime, we send requests to download a specific file concurrently using curl request [246] as much as possible during heavy loads and save the data output metrics for further analysis. To compare the performances between conditions with and without load, we send request to download a specific file repeatedly as much as possible without load. Recently, maximum systems support HTTPS along with HTTP requests for enhancing the security. A simple overhead arises related to HTTPS requests, as it takes more time to download HTTPS type URL for resolving SSL/TLS keys.

Furthermore, concurrent requests may contain same URL or different URL which can also affect performances. Hence, request protocol type and URL type are two necessary metrics to design test case scenarios. As we have two type of servers, cache and backend cloud server, the server type is also a variable metric. Cache hits and misses are other obvious metrics to design test case scenarios for cache servers. Since in case of a cache miss, it takes more time to deliver a response than in the case of a cache hit. To summarize, we state that server type, protocol type, URL type, with or without load conditions, cache hits and misses are our metrics for designing test case scenarios. We propose several test case scenarios varying those parameter metrics such as:
1***) Protocol:*** Request protocol types are of two types i.e. HTTP and HTTPS request protocol.
2***) URL:*** Test cases vary according to request URL (Uniform resource locater). Hence, performance of cache and backend servers depend on the concurrent hits of same or different URL.
3***) Server:*** Backend and cache servers are two parameter metrics for designing the test cases. For cache server, two other important metrics are cache hit and miss based on different URL.
4***) Load:*** For analyzing the system's functionality properly, developers should find out how the system will behave with or without load conditions. Hence, we choose them as important metrics for designing the test case scenarios.

From the combination of these parameters, we design a total of 24 test case scenarios i.e. TC0, TC1, TC2, up to TC23. Table 7.1 and Figure 7.3 presents the test case scenarios. Here, TC4, TC5, TC16, and TC17 are invalid test cases as same URL cache miss can not be possible. Hence, we run load tests for each of the other 20 test cases.

7.3.3 Creating and Disseminating Scripts

We design several shell scripts for determining some metrics e.g. download time, connection time, HTTPS resolve time, etc. We run the scripts to hit URL either in backend or cache, depending

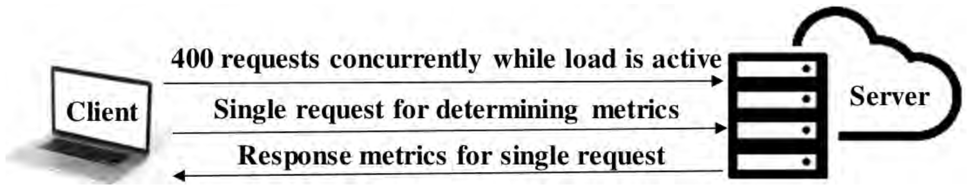

400 requests concurrently while load is active
Single request for determining metrics
Response metrics for single request
Client
Server

Figure 7.4: Architecture of svLoad for single client

on test cases. We also run JMeter [237] in every machine independently. Hence, we have machine independent data for further processing. Furthermore, we design JMeter scripts for all 20 test cases and a bash script to send a specific requests multiple times for measuring performances. We manage the whole task from one management machine using Ansible [238] to install required software, transfer scripts to all machines and run test cases for specified durations.

After running test cases, results are saved to a specific folder in each machine. They are then moved to a management node to merge them for analysis, and converted to a central excel file. The whole task is completed from management node using minimal commands. Since we also need to extract all URL information of user accounts from backend server for concurrent get requests, we design the following script files:

1*)* ***UrlExtractScript:*** Scripts used to extract all URL from backend server. These script were run from management node.
2*)* ***JMeterScript:*** JMeter script for all test cases. This script was run individually from all test client machines.
3*)* ***ResponseMetricsScript:*** This script is used to collect data variables from each curl requests after a complete transfer of requests. This script extracts some predefined data metrics from every URL request and averages the data results. We collect several necessary data metrics among the responses from curl requests. Besides, we find out that these data metrics are important for analyzing system behavior for further improvement. Here, we summarize the critical data variable metrics from curl response [246]:
Size_download: The total amount of bytes that are downloaded. *Size_header:* The total amount of bytes of the downloaded headers. *Size_request:* The total amount of bytes that are sent in the HTTP request. *Speed_download:* The average download speed that curl measured for the complete download. *Time_appconnect:* The time, in seconds, needed from the start until the SSL/SSH/etc connect/handshake to the remote host is completed. *Time_connect:* The time, in seconds, it takes

from the start until the TCP connect to the remote host (or proxy) is completed. *Time_namelookup:* The time, in seconds, it takes from the start until the name resolving is completed. *Time_pretransfer:* The time, in seconds, it takes from the start until the file transfer is just about to begin. *Time_redirect:* The time, in seconds, it takes for all redirection steps including name lookup, connect, pre-transfer and transfer before the final transaction is started. *Time_starttransfer:* The time, in seconds, it takes from the start until the first byte is just about to be transferred. This includes time_pretransfer and also the time the server needs to calculate the result. *Time_total:* The total time, in seconds, that the full operation lasted.
4***) DataAnalysisScript:*** This script moves all files to central management node and generates an excel file from response metrics.

Furthermore, in a distributed denial-of-service (DDoS) attack, multiple compromised computer systems attack a server, website or other network resource. In our proposed methodology, our target is also to flood requests in cache or backend server using proposed test cases and observe miss rate, CPU, and memory usages. From statistics of these usages, we can check stability of our system under DDoS attacks as well.

In summary, JMeter allows maximum 400 to 500 concurrent requests from a single machine. We ran around concurrent 4000 load requests using JMeter from all client machines to the respective cache or backend servers. Besides, we sent a single get request sequentially using our proposed script algorithm for obtaining download related necessary information under this huge load test (in Figure 7.4). We also transfer and collect files and automate the architecture using Ansible tool. This test based architecture using tools JMeter, Ansible and proposed scripts for load testing is not proposed yet in any literature.

7.4 Experimental Evaluation

We evaluate performance of our proposed load test architecture through a real implementation. We also present a comparison of performance between Varnish cache and Swift backend server after tuning the network system, Varnish cache, and Swift backend parameters in real scenarios. Before this, we first elaborate our experimental settings.

Table 7.2: Geographic locations of all machines

Machine name	Machine type	Geographic location
Ba1	Backend Server	Montreal, Canada
Ca1	Cache Server	Montreal, Canada
Ma1	Management Server	Montreal, Canada
A	Client 1	Montreal, Canada
B	Client 2	Montreal, Canada
C	Client 3	Toronto, Canada
D	Client 4	Montreal, Canada
E	Client 5	Toronto, Canada
F	Client 6	Toronto, Canada
G	Client 7	New Jersey, USA
H	Client 8	Montreal, Canada
I	Client 9	Toronto, Canada
J	Client 10	Montreal, Canada

Table 7.3: Configuration of machines used in our load testing

Informations	Backend server	Cache server	Management machine	Client machine
Architecture	x86_64	x86_64	x86_64	x86_64
CPU(s)	16	48	16	1
On-line CPU(s) list	0-15	0-47	0-15	0
Thread(s) per core	2	1	2	1
Core(s) per socket	4	12	4	1
Socket(s)	2	4	2	1
NUMA node(s)	2	8	2	1
CPU family	6	16	6	6
Model name	Intel(R) Xeon(R) CPU E5620 @2.40GHz	AMD Opteron(tm) Processor 6174	Intel(R) Xeon(R) CPU E5620 @2.40GHz	QEMU Virtual CPU version 1.5.3
CPU MHz	2394.141	2199.967	2394.103	2393.998
Virtualization type	VT-x	AMD-V	VT-x	full Storage

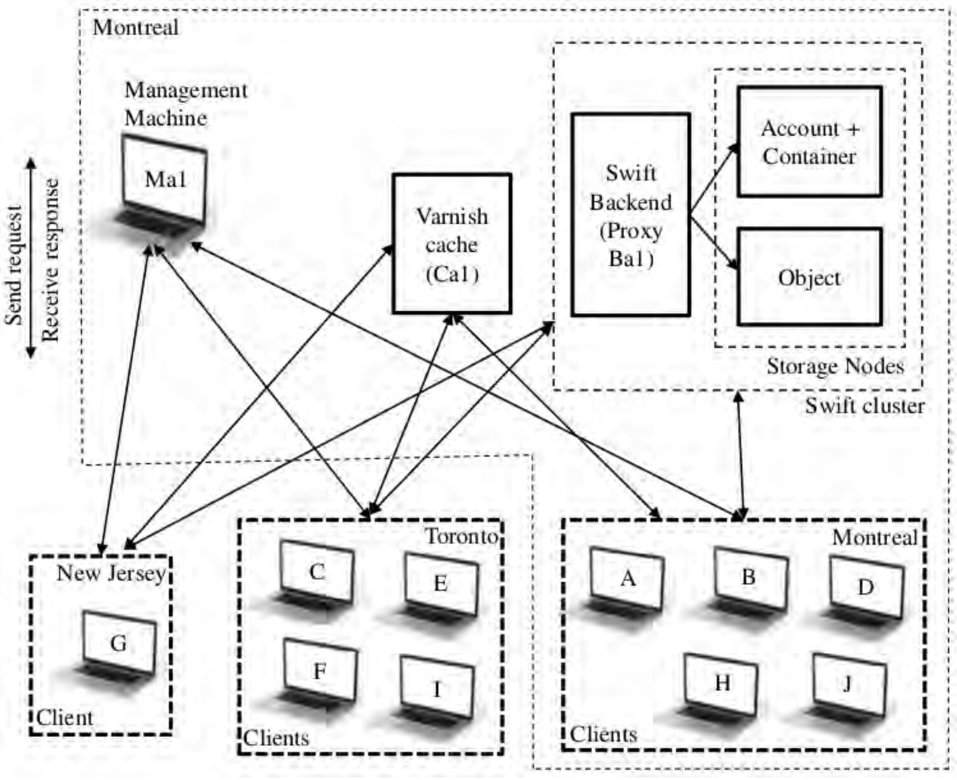


Figure 7.5: Experimental settings of testbed

7.4.1 Experimental Settings

We use state-of-the-art configured machines i.e. one Swift cluster, one Varnish cache, one management machine, and ten client machines which are distributed to three different geographical locations i.e., Montreal and Toronto in Canada, and New Jersey in USA (Table 7.2). Table 7.3 presents the hardware and software related informations of machines used for load testing. The average upload speeds of machines located in Montreal, Toronto, and New Jersey are 5.66 Mbps, 8.64 Mbps, and 207.9 Mbps respectively. Average download speeds are 14.35 Mbps, 3.98 Mbps, and 9.33 Mbps respectively.

Varnish cache server has 16 Gb memory, 64 Gb hard disk and six 1 Gb network interface cards. We install one proxy, one account-container, and one object server for Swift cluster. The memory and disk configurations of Swift servers are as follows: one proxy having 32 Gb memory and 1.2 Tb disk, one account-container having 32 Gb memory and 3 disks each of 400 Gb, and one object having 32 Gb memory and 3 disks each of 400 Gb. Each server had six 1 Gb network interface cards. Figure 7.5 presents the experimental setups of our testbeds. Here, we focus on proxy server as backend server as all requests hit through proxy server for further processing. We create $10,000$ accounts

Table 7.4: Results for TC0 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.019084	60742.78516	0.181213	0.020035	0.181265	0.201807	0.202337
B	0.028005	54005.11328	0.199899	0.029095	0.19996	0.219461	0.220196
C	0.004699	43284.96484	0.241165	0.014015	0.24128	0.267824	0.274486
D	0.016236	50454.20703	0.219792	0.017153	0.219866	0.239778	0.240297
E	0.00502	41436.89453	0.253942	0.014245	0.254066	0.299559	0.306093
F	0.004529	43304.47266	0.24062	0.013669	0.240748	0.267333	0.27393
G	0.00554	44184.625	0.232295	0.016577	0.232384	0.261546	0.26961
H	0.014866	43963.67578	0.250164	0.015741	0.250259	0.269775	0.270343
I	0.00518	40665.79297	0.25983	0.01497	0.259972	0.286211	0.292828
J	0.030641	59550.21094	0.183136	0.03137	0.183174	0.200973	0.201464

and $10,000$ containers in Swift cluster and upload around $55,000$ image files in those accounts for concurrent requests. We use 10 clients to provide concurrent loads on server, and from each client, 400 concurrent requests are sent for each test case.

7.4.2 Experimental Results

In this section, first, we present experimental results. Next, we delineate the parameters for system and network tuning.

Running the Scenario: We run the whole testing process 5 to 6 times with 2 hours duration for each test cases. We needed around 15 to 20 days for collecting the results and tuning the system. We also collect some predefined data response metrics from curl responses i.e., HTTP connection time, dns lookup time, download speed, app connection time, connection time, pre-transfer, start transfer, total response time, etc. to measure performances for each test case [246]. Table 7.4 - Table 7.22 present the testcase results before tuning the system. Moreover, Table 7.23 - Table 7.42 present the testcase results after tuning the system. Additionally, Figure 7.6, Figure 7.7, and Figure 7.8 present the comparison of average testcase results.

Monitoring the Scenario: We monitor and collect the output of CPU and memory usage, disk utilization, process queues, JVM out of memory exceptions, etc. while running test cases in corresponding server. We observed that 70% to 80% memory is used for all 48 cores in Varnish cache due

Table 7.5: Results for TC1 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.025941	44218.06641	0.210628	0.026847	0.210681	0.295597	0.296356
B	0.028007	40094.87891	0.226313	0.030835	0.226381	0.310806	0.312144
C	0.004875	32523.82031	0.278864	0.016339	0.278988	0.370114	0.379062
D	0.02888	37737.80859	0.257613	0.029798	0.257688	0.342154	0.342998
E	0.005202	31722.86719	0.287869	0.015415	0.287953	0.378094	0.387241
F	0.004695	32471.86719	0.278408	0.015789	0.278544	0.371103	0.380134
G	0.005815	33870.49219	0.259371	0.018575	0.259466	0.355652	0.366421
H	0.025132	33845.66406	0.286241	0.025839	0.286297	0.372405	0.373271
I	0.005172	31428.41211	0.29311	0.016017	0.293246	0.382293	0.391595
J	0.036088	43424.09766	0.211516	0.036895	0.211562	0.294506	0.295291

Table 7.6: Results for TC2 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.016666	61303.80859	0.192907	0.01802	0.192965	0.194661	0.195684
B	0.028009	56993.98438	0.20456	0.029003	0.204622	0.205325	0.206063
C	0.004555	43706.6875	0.252417	0.013446	0.252517	0.261421	0.269657
D	0.015336	51187.59766	0.22953	0.016648	0.229614	0.231076	0.23216
E	0.004752	43076.37109	0.25647	0.013766	0.256587	0.265583	0.27392
F	0.004524	44794.70703	0.245845	0.013504	0.24594	0.254876	0.263174
G	0.008603	46317.6875	0.237745	0.019059	0.237822	0.248561	0.258655
H	0.020079	44458.49219	0.266636	0.021554	0.266702	0.268365	0.269538
I	0.004739	42573.92578	0.259644	0.013798	0.259754	0.268843	0.27727
J	0.034873	60438.15234	0.199091	0.036552	0.19913	0.200547	0.201592

Table 7.7: Results for TC3 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.200632	8288.029297	1.472068	0.491215	1.472116	1.651441	1.99448
B	0.028011	10122.91699	1.061443	0.272406	1.061518	1.246851	1.575193
C	0.008518	7939.221191	1.187393	0.27813	1.18751	1.362424	1.690457
D	0.19884	8567.459961	1.417034	0.482272	1.417104	1.59278	1.923229
E	0.00983	7846.32959	1.202856	0.278967	1.202953	1.378434	1.703074
F	0.008206	7962.175781	1.208611	0.29059	1.208725	1.385451	1.71428
G	0.013085	7755.3125	1.199246	0.298278	1.199332	1.397689	1.728491
H	0.193529	7419.634277	1.466874	0.503384	1.466984	1.647716	1.984346
I	0.010136	7754.857422	1.224849	0.300677	1.224939	1.403824	1.736876
J	0.176198	9551.195312	1.327161	0.454495	1.327202	1.503813	1.840092

Table 7.8: Results for TC7 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.032118	528282.0625	0	0.033104	0.033108	0.051514	0.051642
B	0.028008	293970.4375	0	0.037151	0.037165	0.070662	0.071871
C	0.004442	365599.8438	0	0.023067	0.023118	0.062519	0.064248
D	0.040462	536258.5	0	0.041397	0.041432	0.060239	0.060324
E	0.004598	362341.5625	0	0.022558	0.022618	0.063979	0.065797
F	0.00447	364735.125	0	0.023572	0.023623	0.061455	0.063675
G	0.004902	328410.8438	0	0.024676	0.024762	0.06443	0.066232
H	0.038558	526368.3125	0	0.039501	0.039505	0.058656	0.058732
I	0.005307	42833.24219	0.265848	0.032531	0.266014	0.341607	0.34974
J	0.054316	310260.4063	0	0.055128	0.055128	0.076465	0.076535

Table 7.9: Results for TC8 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.01634	793150.625	0	0.017545	0.017553	0.018657	0.018711
B	0.028006	403035.5	0	0.02901	0.029011	0.029043	0.029048
C	0.004241	542713.5625	0	0.013129	0.013243	0.021509	0.021669
D	0.01867	819641.875	0	0.019915	0.019918	0.02098	0.021005
E	0.004355	537195.6875	0	0.013345	0.013416	0.021766	0.021968
F	0.004219	540506	0	0.013219	0.013248	0.021513	0.021775
G	0.004637	463077.7813	0	0.01495	0.015119	0.025573	0.025766
H	0.016679	800007.125	0	0.017859	0.01787	0.01898	0.019023
I	0.004487	528053.625	0	0.013589	0.013639	0.02217	0.022476
J	0.031539	388759.25	0	0.032659	0.032662	0.033766	0.033794

Table 7.10: Results for TC9 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.035501	779714.875	0	0.03955	0.03956	0.041618	0.042822
B	0.031233	365632.7813	0	0.035398	0.035413	0.03876	0.044857
C	0.005066	483237.6875	0	0.019073	0.019181	0.030652	0.036963
D	0.04306	797047.0625	0	0.047339	0.047343	0.050148	0.051527
E	0.005443	479925.0938	0	0.018925	0.019012	0.030551	0.036844
F	0.004677	482887.625	0	0.01913	0.019178	0.03065	0.036974
G	0.004998	417355.4688	0	0.019258	0.019437	0.032841	0.038926
H	0.04442	783647.875	0	0.048976	0.048988	0.051447	0.052671
I	0.00532	474259.5938	0	0.019547	0.019606	0.031274	0.037624
J	0.057037	380339	0	0.062501	0.062507	0.065914	0.067976

Table 7.11: Results for TC10 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.015693	791184	0	0.017008	0.017017	0.018218	0.018355
B	0.028436	401120.625	0	0.029506	0.029508	0.029616	0.029686
C	0.004224	540269.9375	0	0.013374	0.013499	0.021853	0.022126
D	0.015018	812451.875	0	0.016396	0.0164	0.017575	0.017709
E	0.004389	532594.25	0	0.013691	0.013759	0.022222	0.022571
F	0.004252	536894.25	0	0.013624	0.013664	0.022023	0.022366
G	0.004508	462244.4063	0	0.015063	0.01522	0.025754	0.026082
H	0.01524	793786.25	0	0.016703	0.016716	0.017917	0.018047
I	0.004396	529712.6875	0	0.013796	0.01384	0.022363	0.022779
J	0.030542	386226.125	0	0.031954	0.031958	0.033186	0.033347

Table 7.12: Results for TC11 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.019634	721993.6875	0	0.025823	0.025838	0.031075	0.039147
B	0.029113	354770.1563	0	0.03693	0.036943	0.043712	0.056926
C	0.004572	474768.75	0	0.021725	0.021837	0.036444	0.049059
D	0.022678	725800.625	0	0.030354	0.030365	0.036878	0.047431
E	0.005447	471577.6563	0	0.02177	0.021853	0.036596	0.049283
F	0.004729	474586.25	0	0.021748	0.0218	0.03631	0.048698
G	0.004847	415515.6875	0	0.021476	0.021635	0.037804	0.049496
H	0.023517	718484.625	0	0.03138	0.0314	0.037769	0.048055
I	0.006019	466451.1875	0	0.022209	0.02227	0.037181	0.050129
J	0.03905	349520.0313	0	0.047338	0.047348	0.05411	0.06533

Table 7.13: Results for TC12 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.015487	65150.70313	0.165426	0.016397	0.165461	0.182837	0.183459
B	0.028004	54259.375	0.199812	0.029462	0.199873	0.217369	0.218188
C	0.004641	44056.16016	0.237686	0.014099	0.23782	0.263421	0.26915
D	0.015714	48451.64453	0.228081	0.016631	0.228154	0.245271	0.245894
E	0.004814	43232.64063	0.243494	0.014178	0.243625	0.268547	0.274406
F	0.004703	44301.62109	0.235781	0.014175	0.235909	0.2616	0.267554
G	0.005404	47149.33984	0.220683	0.01673	0.220772	0.248511	0.255698
H	0.01869	47133.57422	0.23321	0.019336	0.233289	0.25055	0.251207
I	0.004654	43755.59766	0.239956	0.01418	0.240084	0.265467	0.271249
J	0.029555	60144.29297	0.180714	0.030234	0.180757	0.197452	0.198023

Table 7.14: Results for TC13 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.041417	19062.94531	2.036389	3.376111	2.036472	2.130972	49.295055
B	0.028	29693.16602	0.516917	0.021208	0.516917	0.564125	35.015644
C	0.015971	14744.48535	2.164771	0.5622	2.164829	2.254543	49.066856
D	0.047805	22182.56055	0.955463	2.089927	0.955512	1.011781	43.456512
E	0.033533	12407.09961	0.945433	1.299933	0.9455	1.002933	56.644001
F	0.021182	14687.39356	2.490757	2.880212	2.490849	2.56397	53.642487
G	0.024212	12590.90918	2.097788	0.997849	2.097788	2.170788	51.814091
H	0.472769	11528.07715	1.276885	0.318577	1.276923	1.322808	64.375961
I	0.028	10858.63867	2.319195	0.164028	2.319278	2.415361	49.175667
J	0.184943	21486.14258	2.061286	0.290429	2.061314	2.1414	48.794056

Table 7.15: Results for TC14 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.020115	62194.66797	0.192363	0.021179	0.192394	0.194189	0.195367
B	0.028013	56497.32031	0.205968	0.029005	0.206037	0.207074	0.208006
C	0.004756	43348.76172	0.254602	0.013649	0.254656	0.263959	0.27223
D	0.01754	47940.11328	0.246648	0.01869	0.246721	0.248382	0.249612
E	0.005014	41433.00391	0.267606	0.01408	0.267665	0.277162	0.285528
F	0.004945	43368.67969	0.254148	0.013957	0.254247	0.263593	0.27189
G	0.005551	47390.875	0.227961	0.015854	0.228029	0.239059	0.249047
H	0.020341	44710.39844	0.263499	0.021508	0.263541	0.265465	0.266742
I	0.005045	41742.18359	0.264999	0.014134	0.265111	0.274604	0.283046
J	0.032237	62011.65625	0.191228	0.033577	0.191266	0.192807	0.193995

Table 7.16: Results for TC15 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.340213	8257.826172	1.36538	0.531221	1.365437	1.531829	1.830516
B	0.034971	12095.40039	0.770803	0.17654	0.770899	0.914825	1.181145
C	0.027634	8160.665039	1.055008	0.185498	1.05425	1.226756	1.506746
D	0.289014	10514.98438	1.167743	0.462584	1.167833	1.329151	1.609664
E	0.027378	8218.579102	1.059135	0.185662	1.058348	1.225625	1.509194
F	0.022364	8316.642578	1.022973	0.187677	1.023064	1.192021	1.476339
G	0.020126	8962.051758	0.934746	0.179293	0.934823	1.098645	1.380445
H	0.441911	6006.353516	2.069026	0.645061	2.069109	2.283316	2.605666
I	0.039189	7751.248047	1.136165	0.210912	1.136281	1.314614	1.601192
J	0.244803	11135.72363	1.078655	0.426397	1.078705	1.234622	1.517611

Table 7.17: Results for TC18 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.019293	60225.92188	0.180424	0.020203	0.180472	0.200928	0.202637
B	0.028007	53157.89063	0.201172	0.029367	0.201236	0.221682	0.223546
C	0.004801	42564.10547	0.242631	0.014366	0.242767	0.271225	0.279649
D	0.014409	46712.40234	0.232744	0.015332	0.232821	0.252631	0.254377
E	0.00479	42623.89063	0.243128	0.013958	0.24325	0.270935	0.279285
F	0.004605	43994.09375	0.233621	0.013787	0.233751	0.261823	0.27013
G	0.005396	46249.87891	0.216528	0.016403	0.216617	0.247287	0.257283
H	0.024845	46774.11719	0.240828	0.025588	0.240914	0.261036	0.262604
I	0.004869	42543.89063	0.242976	0.014103	0.243102	0.270995	0.279492
J	0.032789	58923.96875	0.185032	0.03347	0.185072	0.204127	0.205782

Table 7.18: Results for TC19 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.022756	32821.96484	0.420519	0.054385	0.420473	0.563559	0.65878
B	0.028035	31139.11719	0.439933	0.04509	0.440003	0.58818	0.687048
C	0.011461	24325.25586	0.490483	0.031592	0.490622	0.644817	0.828277
D	0.019447	28145.87891	0.473768	0.053924	0.473855	0.623012	0.726451
E	0.011898	24212.84961	0.553785	0.054419	0.553932	0.712399	0.830077
F	0.006669	26799.12109	0.445243	0.024845	0.44537	0.60379	0.771398
G	0.012083	27614.77344	0.49808	0.075417	0.498157	0.650564	0.75882
H	0.034107	23120.46289	0.578881	0.093225	0.578979	0.738059	0.857449
I	0.007579	26166.95508	0.505028	0.046531	0.505162	0.663324	0.774317
J	0.030267	34310.60156	0.410355	0.059561	0.410398	0.553371	0.6474

Table 7.19: Results for TC20 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.014341	58309.9375	0.200236	0.015643	0.200278	0.202193	0.203155
B	0.028	57137.01563	0.203484	0.028994	0.203511	0.204654	0.205273
C	0.004954	44078.00781	0.2506	0.013957	0.250719	0.260073	0.268345
D	0.015503	48502.86719	0.241705	0.016509	0.241774	0.243328	0.244321
E	0.00519	42452.80859	0.26077	0.014328	0.260834	0.270235	0.278604
F	0.004716	45191.86328	0.243213	0.013777	0.243311	0.252698	0.261034
G	0.006003	48840.25391	0.220463	0.01635	0.220527	0.231495	0.241529
H	0.018507	45689.35938	0.258577	0.019776	0.258653	0.260429	0.261422
I	0.005427	41473.03906	0.267132	0.01468	0.267256	0.276866	0.285513
J	0.030655	62132.97656	0.18865	0.031407	0.188692	0.19024	0.1912

Table 7.20: Results for TC21 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.068964	55690.34375	0.291737	0.082238	0.291787	0.300865	0.312276
B	0.04071	48872.01953	0.249366	0.04903	0.249433	0.258767	0.273809
C	0.012136	37919.80078	0.331486	0.031305	0.331602	0.349638	0.367839
D	0.093904	38641.80078	0.381017	0.108879	0.381109	0.391742	0.407276
E	0.011802	36455.53125	0.336256	0.028871	0.336372	0.354912	0.374287
F	0.01087	37726.29297	0.326415	0.030176	0.32652	0.346393	0.366749
G	0.012572	40000.26563	0.308993	0.032168	0.309062	0.330024	0.350814
H	0.086221	37405.98047	0.426654	0.102479	0.426682	0.43697	0.449293
I	0.021803	35802.99609	0.353976	0.040418	0.354098	0.373446	0.393639
J	0.071201	53370.94531	0.272788	0.086979	0.272828	0.283183	0.296794

Table 7.21: Results for TC22 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.016235	62523.93359	0.188493	0.017258	0.188548	0.190152	0.190908
B	0.028025	57322.53906	0.203324	0.029013	0.203364	0.204289	0.2049
C	0.004918	42840.45313	0.258386	0.01391	0.258495	0.267569	0.275848
D	0.018143	48220.06641	0.245875	0.019264	0.245946	0.247225	0.248126
E	0.005063	43041.34375	0.257357	0.014125	0.257471	0.266589	0.27494
F	0.004793	45082.66016	0.244508	0.01383	0.24462	0.253679	0.262001
G	0.006339	47189.92188	0.229855	0.016726	0.22989	0.240695	0.250738
H	0.02028	45725.02734	0.260411	0.02123	0.260492	0.262018	0.262962
I	0.012978	41760.20703	0.272946	0.022405	0.273014	0.282621	0.29142
J	0.032245	62336.39453	0.190522	0.033311	0.190562	0.191841	0.19267

Table 7.22: Results for TC23 before tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.032456	58487.58594	0.238117	0.040948	0.238175	0.244805	0.254168
B	0.04302	51359.41016	0.242575	0.049136	0.242628	0.24959	0.261084
C	0.011445	39702.35938	0.330751	0.033481	0.330871	0.347802	0.365594
D	0.02911	43205.52344	0.293325	0.04106	0.293395	0.301226	0.313982
E	0.008601	39221.67188	0.30985	0.026888	0.309957	0.326395	0.348524
F	0.009166	40093.44922	0.315406	0.031505	0.315463	0.331211	0.347473
G	0.008819	41766.10156	0.279098	0.029201	0.279176	0.29651	0.316196
H	0.034602	44148.99609	0.305516	0.045171	0.305604	0.312865	0.322424
I	0.011835	37385.64063	0.335938	0.030722	0.336067	0.352669	0.376666
J	0.044108	56492.02734	0.230668	0.054705	0.230714	0.237998	0.249817

Table 7.23: Results for TC0 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.10765	299715.0625	0	0.10774	0.107743	0.116995	0.117044
B	0.028007	303402.4375	0	0.029666	0.029673	0.039327	0.039455
C	0.039049	209277.9688	0	0.048028	0.048137	0.066085	0.066308
D	0.096488	305289.9063	0	0.097115	0.097116	0.106006	0.10607
E	0.040415	204297.125	0	0.049519	0.049687	0.067847	0.068296
F	0.042825	209526.125	0	0.052306	0.052434	0.070326	0.070588
G	0.05625	197519.0313	0	0.067622	0.067663	0.087348	0.087627
H	0.094761	301850.4063	0	0.094821	0.094822	0.104054	0.104111
I	0.034882	40178.30469	0.273846	0.044552	0.273989	0.293171	0.299291
J	0.032298	311329.6875	0	0.033161	0.033185	0.041944	0.041999

Table 7.24: Results for TC1 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.067637	145203.6875	0	0.067831	0.067837	0.125331	0.125383
B	0.028009	145929.1875	0	0.029539	0.029562	0.089791	0.09001
C	0.039765	118106.8594	0	0.04956	0.049679	0.118349	0.118649
D	0.061503	145939.3906	0	0.062224	0.062221	0.121291	0.121366
E	0.04205	115116.9375	0	0.052436	0.052595	0.11983	0.120133
F	0.042369	117443.1563	0	0.052442	0.052565	0.120164	0.120453
G	0.044978	114642.3594	0	0.056354	0.056392	0.127184	0.127475
H	0.059853	143400.4375	0	0.059905	0.059911	0.119873	0.119946
I	0.034539	31739.08398	0.304247	0.045303	0.304393	0.369301	0.378297
J	0.034022	149101.1875	0	0.034971	0.034984	0.094316	0.094377

Table 7.25: Results for TC2 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.116203	368656.1875	0	0.117548	0.117647	0.118984	0.119065
B	0.02801	401086.4375	0	0.029002	0.029003	0.029019	0.029046
C	0.039732	252284.1875	0	0.048287	0.04835	0.057011	0.057301
D	0.104565	374345.5938	0	0.105981	0.106103	0.107341	0.107514
E	0.040403	246212.7344	0	0.04921	0.049322	0.058276	0.058566
F	0.042676	249357.9375	0	0.051287	0.051349	0.06012	0.060389
G	0.067534	235360.8125	0	0.078744	0.078808	0.089003	0.089184
H	0.085421	371751.2188	0	0.086837	0.086939	0.08822	0.088297
I	0.044008	238947.6719	0	0.054079	0.054224	0.063682	0.063994
J	0.030642	382722.1875	0	0.032001	0.032072	0.033333	0.033471

Table 7.26: Results for TC3 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.146231	20884.95508	0	0.427179	0.427258	0.648401	1.001198
B	0.028009	35605.76563	0	0.249395	0.249489	0.447604	0.766879
C	0.03911	17577.25391	0	0.312375	0.312513	0.539352	0.887114
D	0.147004	29269.55664	0	0.443464	0.443536	0.661681	1.004843
E	0.044525	17732.80273	0	0.302281	0.30242	0.534231	0.88254
F	0.041005	17342.20508	0	0.310395	0.310512	0.545381	0.896785
G	0.036223	19518.44531	0	0.304416	0.304496	0.524989	0.864168
H	0.168334	18560.92578	0	0.444694	0.444815	0.664194	1.019172
I	0.076361	16309.88086	0	0.361308	0.36146	0.611714	0.97024
J	0.122749	35183.44922	0	0.411043	0.411097	0.627534	0.968778

Table 7.27: Results for TC6 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.122962	298700.0313	0	0.123179	0.1232	0.133059	0.133141
B	0.028006	302500.8125	0	0.029867	0.029903	0.040155	0.040306
C	0.035432	210188.2813	0	0.045173	0.045271	0.063434	0.063812
D	0.088303	305859.375	0	0.089006	0.089016	0.098406	0.09847
E	0.033706	206180.9688	0	0.043384	0.043541	0.061963	0.062299
F	0.045874	209175.0313	0	0.055513	0.055651	0.073926	0.074258
G	0.073808	197484.8125	0	0.085548	0.085587	0.105414	0.105684
H	0.078407	302724.4063	0	0.078563	0.078585	0.088393	0.088449
I	0.031144	40812.83984	0.268727	0.040492	0.268867	0.287963	0.291605
J	0.031761	311123	0	0.032771	0.032789	0.041853	0.041924

Table 7.28: Results for TC7 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	1.67896	49763.36328	0	1.706753	1.706809	1.941898	2.008173
B	0.028024	58926.67578	0	0.246364	0.246457	0.513585	0.659237
C	0.040843	44555.34766	0	0.282338	0.282466	0.56904	0.689125
D	1.499186	63897.85938	0	1.522197	1.522277	1.723761	1.769869
E	0.032921	42916.48828	0	0.290343	0.290501	0.590388	0.734961
F	0.032915	43657.5625	0	0.258	0.258117	0.521686	0.649974
G	0.046218	41702.34375	0	0.304511	0.304579	0.589784	0.708805
H	1.496822	50696.59375	0	1.525238	1.525311	1.735355	1.789202
I	0.037667	13996.31738	0.879221	0.25594	0.879358	1.257474	1.341999
J	1.247882	78833.97656	0	1.271957	1.272009	1.458364	1.534731

Table 7.29: Results for TC8 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.121152	317083.4063	0	0.126362	0.12642	0.129896	0.130925
B	0.028007	357492.4375	0	0.030598	0.030664	0.032629	0.033117
C	0.041317	221385.7656	0	0.051658	0.051771	0.062223	0.065592
D	0.098539	322316.5313	0	0.102699	0.102767	0.10608	0.107098
E	0.042338	217506.1719	0	0.052928	0.05306	0.06369	0.067101
F	0.041741	221004.3438	0	0.052133	0.052244	0.062685	0.066075
G	0.069422	207286.4844	0	0.081606	0.081687	0.09394	0.097337
H	0.085646	317117.375	0	0.090281	0.090353	0.093861	0.094908
I	0.040539	218907.75	0	0.051188	0.05131	0.061898	0.065299
J	0.037918	330787.0313	0	0.043065	0.043112	0.046336	0.047362

Table 7.30: Results for TC9 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.098253	20289.54102	0	0.265381	0.265462	0.427985	0.74953
B	0.033502	28759.43164	0	0.189196	0.189292	0.34482	0.6546
C	0.041264	20311.50977	0	0.213177	0.213307	0.381978	0.701197
D	0.082821	20711.39844	0	0.25351	0.253581	0.416711	0.738841
E	0.041839	19842.51953	0	0.212823	0.212961	0.383283	0.703608
F	0.032192	19807.03125	0	0.202582	0.20272	0.372227	0.693343
G	0.032553	20552.39844	0	0.204543	0.204625	0.374911	0.694146
H	0.094832	20246.43164	0	0.264151	0.264237	0.42907	0.750191
I	0.043163	19454.21094	0	0.216397	0.216535	0.387461	0.70931
J	0.09828	21789.10742	0	0.266704	0.266771	0.428656	0.750028

Table 7.31: Results for TC10 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.109053	310066.3438	0	0.114703	0.114764	0.118507	0.119811
B	0.028009	360215.25	0	0.030494	0.030559	0.032412	0.032861
C	0.045282	222911.0781	0	0.055466	0.055576	0.0659	0.069109
D	0.109091	316136.125	0	0.114804	0.114869	0.118306	0.119398
E	0.048539	217243.4063	0	0.059087	0.059221	0.069819	0.073001
F	0.041192	222510.4844	0	0.051459	0.051574	0.061928	0.065059
G	0.077574	210076.7813	0	0.089679	0.089758	0.101769	0.104879
H	0.094859	310203.7813	0	0.100064	0.100136	0.103813	0.105048
I	0.045027	220924.6094	0	0.055358	0.055478	0.065903	0.069094
J	0.050513	325287.9063	0	0.056423	0.05647	0.059906	0.061035

Table 7.32: Results for TC11 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.17411	19115.00977	0	0.36944	0.369524	0.544364	0.877221
B	0.040811	20336.81055	0	0.198448	0.198554	0.35551	0.66927
C	0.040568	18661.69727	0	0.218807	0.218941	0.387952	0.706972
D	0.247955	19564.65039	0	0.430383	0.430455	0.59688	0.918507
E	0.058938	18110.55273	0	0.229887	0.23004	0.400459	0.720509
F	0.047221	18554.41016	0	0.228322	0.228449	0.396933	0.716511
G	0.03774	18407.44727	0	0.209022	0.209109	0.380037	0.700079
H	0.238769	17983.19922	0	0.429372	0.429478	0.61019	0.940805
I	0.049686	18344.14844	0	0.219779	0.21992	0.389719	0.719838
J	0.184146	20149.44141	0	0.363277	0.363335	0.526633	0.846268

Table 7.33: Results for TC12 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.046509	59324.42969	0.195983	0.046603	0.196041	0.213169	0.213805
B	0.030416	54052.91406	0.203737	0.032058	0.203799	0.222387	0.223231
C	0.034157	41076.43359	0.260509	0.043517	0.260635	0.286311	0.293313
D	0.038341	43991.16016	0.258883	0.0388	0.258958	0.27519	0.275748
E	0.033781	37778.83203	0.285702	0.043257	0.285855	0.314414	0.321503
F	0.034308	40537.43359	0.266227	0.045239	0.26635	0.291825	0.29896
G	0.028588	44458.07813	0.231495	0.04012	0.231573	0.258673	0.266938
H	0.045345	46533.58984	0.248955	0.045471	0.249029	0.266138	0.266747
I	0.033376	38022.46875	0.28374	0.043295	0.283893	0.308942	0.316035
J	0.032407	59665.14453	0.186089	0.03324	0.18613	0.201693	0.202251

Table 7.34: Results for TC13 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.038	8491.333008	0.599611	3.530944	0.599667	0.666	66.779945
B	0.028	15219.26074	0.584348	2.765261	0.584348	0.646304	52.331654
C	0.045778	7516.944336	0.7935	0.035278	0.7935	0.857167	64.468613
D	0.03592	14342.91992	4.14224	2.70924	4.14228	4.215	49.243999
E	0.04345	7119.299805	2.10545	7.10665	2.10565	2.1839	63.258598
F	0.042	7435.388672	2.345667	0.368056	2.345667	2.448056	64.50322
G	0.03975	7346.350098	1.0473	3.3987	1.0473	1.11985	58.862949
H	0.049095	8922.428711	1.422286	0.171429	1.422476	1.496286	56.023903
I	0.048778	5274.666504	7.635334	5.275667	7.635389	7.758056	66.93589
J	0.028034	13864.31055	5.309345	1.680172	5.309345	5.442207	43.481346

Table 7.35: Results for TC14 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.047081	60383.50781	0.208612	0.048339	0.208667	0.210799	0.212028
B	0.028006	55999.375	0.207801	0.029	0.207861	0.209077	0.209956
C	0.044569	40603.31641	0.286762	0.053037	0.286858	0.296194	0.304425
D	0.047634	43580.29297	0.285038	0.048982	0.285102	0.286907	0.288204
E	0.039416	40343.89453	0.28391	0.047913	0.284014	0.293353	0.301616
F	0.039358	40846.46094	0.280156	0.047868	0.280256	0.289728	0.298017
G	0.031337	45470.62891	0.240585	0.04177	0.24064	0.251664	0.261638
H	0.037871	44804.20703	0.267458	0.039357	0.267541	0.269753	0.271024
I	0.041897	38727.53906	0.298022	0.050516	0.298137	0.30757	0.315926
J	0.034492	59667.71094	0.201569	0.036854	0.201613	0.203473	0.20469

Table 7.36: Results for TC15 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.35707	7803.44043	1.412992	0.576727	1.413062	1.577758	1.87939
B	0.028016	12360.51758	0.783515	0.181574	0.783585	0.923822	1.188147
C	0.086548	7777.142578	1.100523	0.261093	1.100631	1.265544	1.555208
D	0.439214	9289.480469	1.342854	0.627902	1.342925	1.499444	1.782461
E	0.09305	7701.704102	1.117403	0.259847	1.117503	1.287387	1.578319
F	0.082823	8205.996094	1.063078	0.252081	1.063172	1.226385	1.507088
G	0.048291	8908.886719	0.951703	0.214641	0.951776	1.113022	1.396486
I	0.086838	7970.237793	1.101108	0.250776	1.101217	1.268172	1.5534
J	0.33687	10613.62598	1.214724	0.544958	1.214772	1.375362	1.650916

Table 7.37: Results for TC18 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.044639	57777.29297	0.199848	0.044795	0.199905	0.217248	0.21818
B	0.028006	53953.33203	0.201666	0.030277	0.201728	0.219097	0.220068
C	0.033054	38811.60156	0.274838	0.042663	0.274965	0.300516	0.308592
D	0.030923	43428.54297	0.255092	0.031528	0.255162	0.272638	0.273538
E	0.032797	39234.98047	0.271352	0.042438	0.271473	0.297152	0.305357
F	0.033114	39242.30078	0.270677	0.042744	0.270801	0.296887	0.30534
G	0.031502	42824.78125	0.242036	0.042309	0.242118	0.26954	0.279485
I	0.038657	38076.58594	0.284776	0.047611	0.28491	0.311073	0.319201
J	0.031169	58410.91406	0.187668	0.032048	0.187717	0.204357	0.205281

Table 7.38: Results for TC19 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	1.815939	7984.039551	2.47743	1.826362	2.477499	2.949667	3.274169
B	0.028042	9743.496094	1.068403	0.331369	1.068485	1.56011	2.010433
C	0.075415	7761.455078	1.266104	0.320756	1.266208	1.769846	2.175009
D	2.22124	10353.21289	2.755175	2.221749	2.755261	3.254259	3.536749
E	0.085086	7540.192383	1.261645	0.337047	1.261764	1.771196	2.193308
F	0.05637	7319.6875	1.213675	0.307117	1.213769	1.737784	2.172541
G	0.050893	7952.11084	1.185294	0.30274	1.185367	1.706245	2.140634
I	0.108906	7405.003906	1.400635	0.397145	1.40075	1.925076	2.372691
J	1.605437	11368.0459	2.097303	1.606558	2.097339	2.554706	2.92649

Table 7.39: Results for TC20 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.057733	52590.73828	0.242454	0.059412	0.242518	0.245388	0.247385
B	0.028015	55625.87109	0.208716	0.029007	0.208786	0.210272	0.211366
C	0.043405	38375.66797	0.301444	0.052007	0.301553	0.311235	0.319555
D	0.046675	44012.375	0.280012	0.048432	0.280038	0.282457	0.284235
E	0.044299	38735.38281	0.299418	0.052961	0.299448	0.309213	0.317485
F	0.046672	39293.33984	0.297359	0.055274	0.297458	0.307017	0.315345
G	0.033059	43397.3125	0.258155	0.047159	0.258227	0.269552	0.279592
I	0.0579	35526.03906	0.339059	0.069975	0.339183	0.349865	0.358972
J	0.030094	57875.92969	0.202875	0.03289	0.202918	0.205184	0.206927

Table 7.40: Results for TC21 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

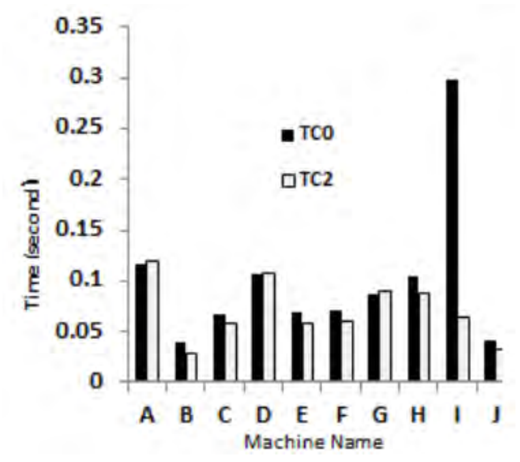
ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.894368	6970.936523	2.141484	1.138863	2.141538	2.351189	2.694182
B	0.033189	12356.10645	0.769806	0.16748	0.769878	0.916205	1.1912
C	0.102406	7579.007324	1.168664	0.257739	1.168757	1.341535	1.635483
D	0.639094	8022.671875	1.6413	0.845463	1.641397	1.83482	2.176845
E	0.077719	7801.708496	1.143136	0.231843	1.143241	1.315901	1.605047
F	0.088172	7754.935059	1.139159	0.242088	1.139275	1.310096	1.602945
G	0.045875	8900.506836	0.99461	0.201546	0.99468	1.162849	1.447641
I	0.356673	6354.593262	1.631673	0.547038	1.631774	1.843522	2.138974
J	0.516642	8869.533203	1.452288	0.7241	1.452345	1.642395	1.964673

Table 7.41: Results for TC22 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

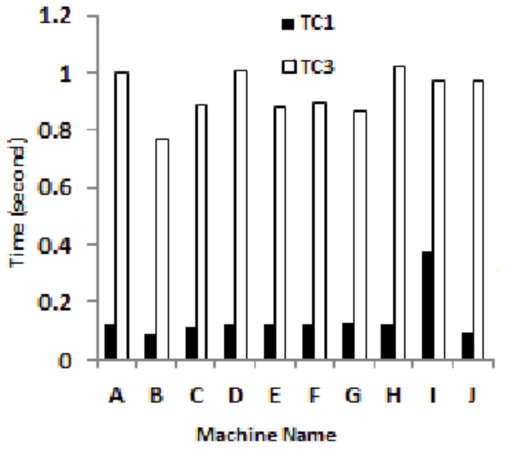
ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.063536	53110.67969	0.245477	0.066303	0.24554	0.248158	0.249992
B	0.028014	55510.45313	0.208884	0.029007	0.208957	0.210549	0.211867
C	0.033416	39147.69531	0.286105	0.041948	0.286219	0.296087	0.30439
D	0.059045	42649.90625	0.295486	0.059841	0.295563	0.297918	0.299696
E	0.045436	38713.27344	0.301409	0.053995	0.301506	0.311222	0.319542
F	0.034773	38861.05859	0.289704	0.043332	0.28981	0.299603	0.307894
G	0.033258	43369.83984	0.254754	0.043714	0.254825	0.266222	0.276288
I	0.056157	35340.35938	0.337512	0.065862	0.337635	0.348377	0.35724
J	0.037149	57498.22266	0.210251	0.038759	0.210295	0.212711	0.214395

Table 7.42: Results for TC23 after tuning. Here, we use header size - 397 byte, size download - 11742 byte, and request size - 124 byte. All the sizes and time are represented in byte and seconds respectively.

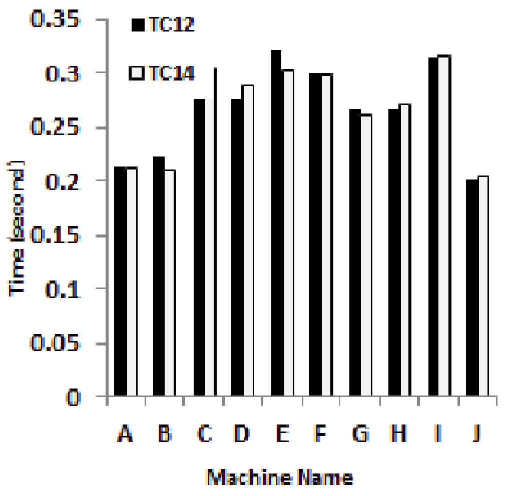
ID	Lookup time	Download speed	App conn time	Conn time	Pre transfer	Start transfer	Time
A	0.301887	7322.029297	1.388417	0.47247	1.388477	1.566566	1.886102
B	0.033683	10623.19531	0.813872	0.18062	0.813957	0.973608	1.269982
C	0.07948	7473.799316	1.15516	0.246792	1.155246	1.335187	1.644288
D	0.213517	9621.334961	1.069908	0.377284	1.06999	1.235521	1.53714
E	0.077606	7577.475586	1.137657	0.242391	1.137762	1.318299	1.626053
F	0.066898	7676.483398	1.112433	0.232253	1.112535	1.289089	1.595072
G	0.042545	8334.337891	0.988497	0.205532	0.988571	1.164895	1.471552
I	0.102632	6709.590332	1.331898	0.274095	1.331996	1.523117	1.830178
J	0.101291	10405.08106	0.924151	0.261743	0.924199	1.091777	1.393926



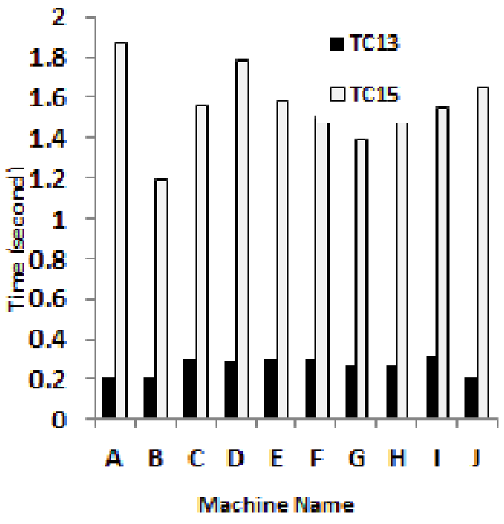
(a) TC0 vs TC2



(b) TC1 vs TC3

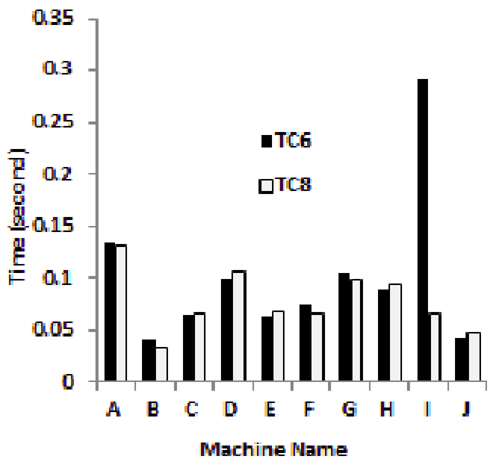


(c) TC12 vs TC14

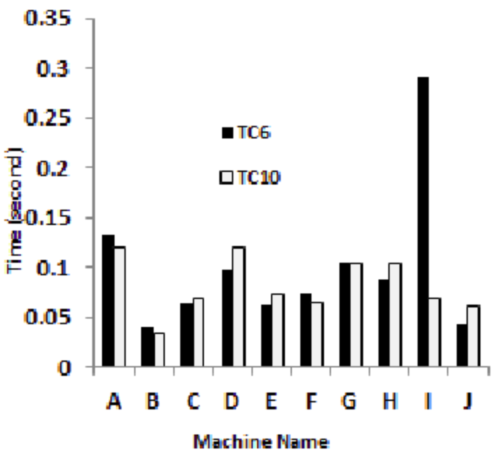


(d) TC13 vs TC15

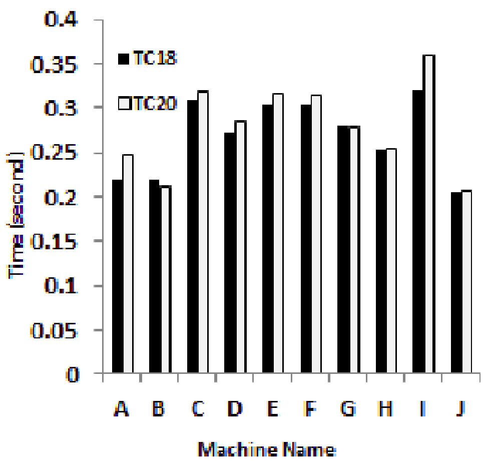
Figure 7.6: Testcase results - case 1



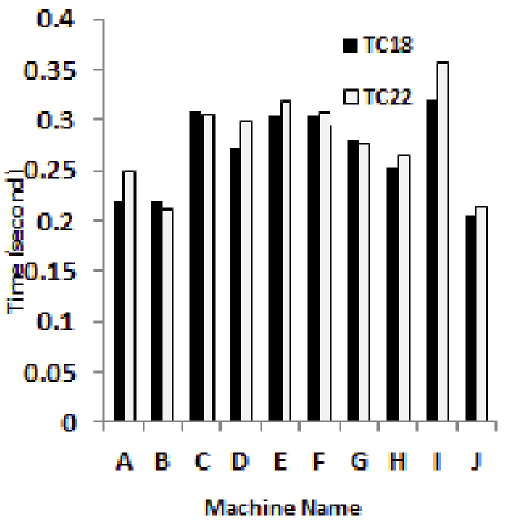
(a) TC6 vs TC8



(b) TC6 vs T10

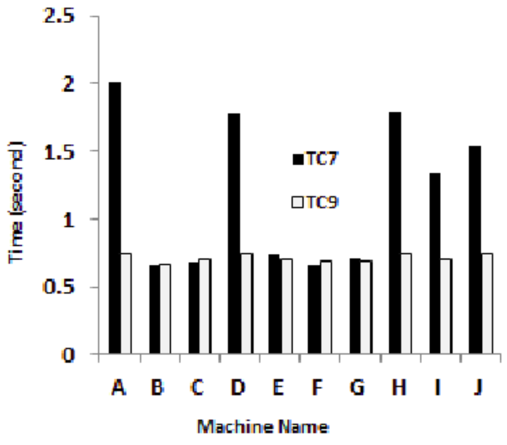


(c) TC18 vs TC20

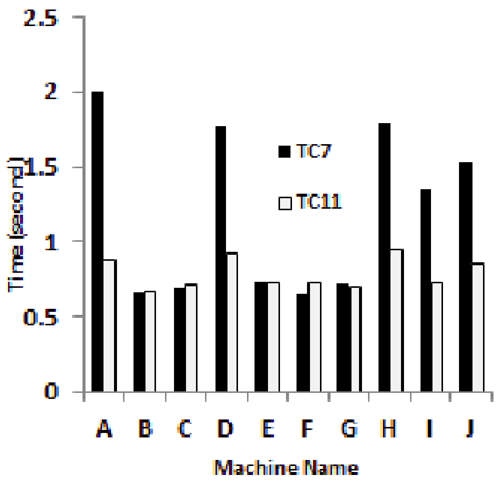


(d) TC18 vs TC22

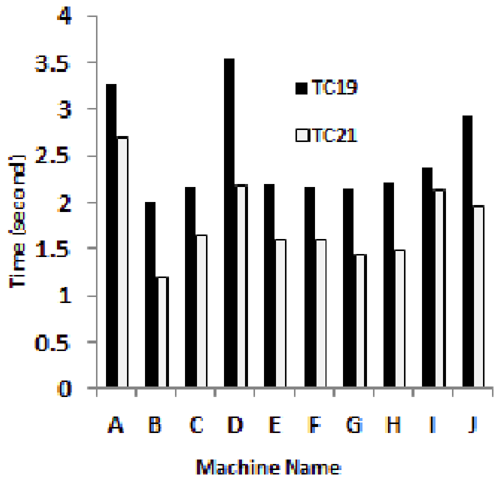
Figure 7.7: Testcase results - case 2



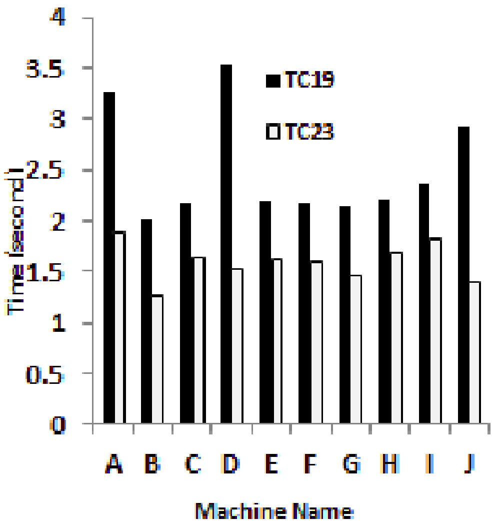
(a) TC7 vs TC9



(b) TC7 vs T11



(c) TC19 vs TC21



(d) TC19 vs TC23

Figure 7.8: Testcase results - case 3

Table 7.43: Varnish cache server tuning parameters

Parameters	Default values	Intermediate tuned values	Final tuned values
Thread_pool_minimum	5	5	400
Thread_pool_maximum	500	1000	5000
Thread_queue_limit	20	50	100
Workspace_thread	2k	4k	8k
Workspace_session	4k	0.5k	0.5k
Pipe_timeout	60 sec	15 sec	30 sec
Lru_interval	2 sec	10 sec	20 sec
Listen_depth	1024	2048	4096

Table 7.44: Swift backend server tuning parameters

Parameters	Variables	Default values	Intermediate tuned values	Final tuned values
Memcached	MAXCONN	1024	2048	4096
	CACHESIZE	64k	1024k	4096k
File Descriptor	fs.file-max	8192	32768	2097192
Ulimit	Hard limit	4096	100000	400000
	Soft limit	1024	4096	100000

Table 7.45: System network tuning parameters

Parameters	Default values	Intermediate tuned values	Final tuned values
net.core.wmem_default	212992	131072	262144
net.core.wmem_max	212992	1048576	4194304
net.core.rmem_default	212992	131072	262144
net.core.rmem_max	212992	1048576	4194304

to leveraging tasks to OS. After running 1st round of load test, we observe that some parameters of Swift, Varnish, and machine's network system must be tuned for better performance. We find out several bottlenecks related to these software through continuous load testing.

Next, we present necessary components and proper parameter values related to system and network tuning from analysis of data metrics through rigorous load testing. We benchmark system behavior through changing default values gradually identifying system metrics. Here, we only present best tuning values due to lack of space.

Table 7.46: Success and miss rates (%) of requests

Test case id	Before Tuning		After Tuning	
	Success	Miss	Success	Miss
TC13	57%	43%	100%	0%
TC15	99%	1%	100%	0%
TC19	51%	49%	99%	1%
TC21	99%	1%	100%	0%
TC23	99%	1%	100%	0%

Swift Tuning: We locate three bottlenecks related to Swift tuning. When maximum load is given to the backend Swift proxy server, the memory cache (*memcached*) fails to handle large amount of requests, and after sometime, unsuccessful responses are generated. So, we change the *memcache.conf* file and increase the size of memory, cache and maximum number of allowed connection, and executed the load tests again. We also find out some parameters related to *memcache*, file descriptor, and ulimit that have great impact on load tests by changing them repeatedly. Table 7.44 presents the default and tuned values for Swift.

Varnish Tuning: We tune the necessary parameters of Varnish cache i.e. Thread_pool_minimum, Thread_pool_maximum, Thread_queue_limit, Workspace_thread, Workspace_session, Pipe_timeout, Lru_interval and Listen_depth, keeping other values default. Table 7.43 presents default and tuned values for Varnish cache.

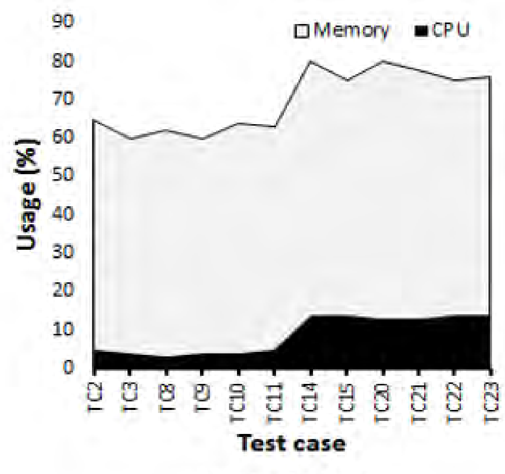
Network system tuning: Kernel buffer parameters i.e. net.core.wmem_default, net.core.rmem_default, net.core.rmem_max and net.core.wmem_max show the default and maximum write (receiving) and read (sending) buffer size allocated to any type of connection. The default values are low since the allocated space is taken from the RAM. Increasing this improves the performance for systems running servers. Table 7.45 presents the defaults and tuned values for network system.

After 1st round of load test, we observe that for TC0, TC1, TC2, TC3, TC6, TC7, TC8, TC9, TC10, TC11, TC12, TC14, TC18, TC20, and TC22 success response is 100%. Five test cases have highest miss rate due to HTTPS requests and lack of tuning. Hence, we tune the system and perform load test multiple times. After tuning, success rates for all test cases improved to 99%. We present the success and miss rates for TC13, TC15, TC19, TC21, and TC23 in Table 7.46.

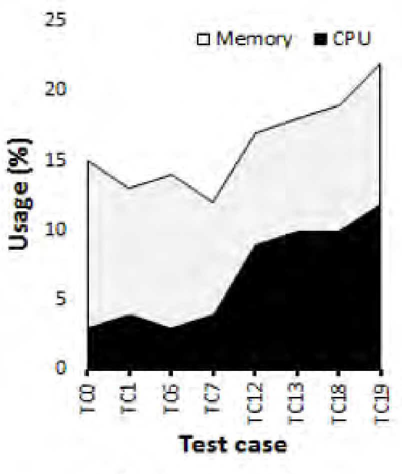
Table 7.47 shows the %improvement of response time after final tuning. In these test cases, miss rate

Table 7.47: Percentage of average response time improvement for all clients

Test case id	Response time (s)		Improvement
	Before tuning	After tuning	
TC0	0.10	0.02	79%
TC1	0.14	0.41	-66%
TC2	0.07	0.03	57%
TC3	0.93	1.13	-18%
TC6	0.10	0.02	81%
TC7	1.20	0.47	60%
TC8	0.08	0.02	69%
TC9	0.71	0.42	41%
TC10	0.08	0.02	69%
TC11	0.78	0.82	-5%
TC12	0.27	0.24	10%
TC13	0.27	1.83	-85%
TC14	0.27	0.25	6%
TC15	1.56	2.14	-27%
TC18	0.27	0.24	10%
TC19	2.50	1.66	34%
TC20	0.28	0.25	10%
TC21	1.79	1.18	34%
TC22	0.28	0.25	12%
TC23	1.59	1.67	-5%



(a) Cache server



(b) Backend server

Figure 7.9: Average memory and CPU usage of cache and backend server for 20 test cases

Table 7.48: Improvement of cache server response times over backend server

Backend Test case id	Cache Test case id	Improvement (%)
TC0	TC2	29%
TC1	TC3	-85%
TC6	TC8	23%
	TC10	18%
TC7	TC9	40%
	TC11	34%
TC12	TC14	0%
TC13	TC15	-83%
TC18	TC20	-4%
	TC22	-4%
TC19	TC21	29%
	TC23	36%

Table 7.49: Comparison of HTTP request response times over HTTPS request response times

HTTP Test case id	HTTPS Test case id	Improvement on HTTP (%)
TC0	TC12	91%
TC1	TC13	77%
TC2	TC14	87%
TC3	TC15	47%
TC6	TC18	91%
TC7	TC19	71%
TC8	TC20	90%
TC9	TC21	64%
TC10	TC22	90%
TC11	TC23	50%

is high hence total response time is bit higher before tuning. Furthermore, after tuning the system, miss rate is decreased hence lower the response time. Figure 7.9 presents average memory and CPU usage of cache and backend server. These usage are remain almost same after the tuning.

7.4.3 Experimental Findings

In this section, we present the findings of tuning the system, cache and backend, and comparing request times of HTTP and HTTPS by analyzing the results.

Analyze test results: We analyze test cases for comparing the average response time from all the clients for Varnish cache and Swift backend server. This behavior remains same after tuning the servers as response time ratio from cache and backend servers varies depending on the test cases. Note that, each client machine requests for downloading same URL once when testing without load criterion, hence clients hit total 10 requests concurrently. Besides, each client machines requests for downloading same or different URL for 400 times when testing with load criterion, hence clients hit total 4000 requests concurrently.

Furthermore, we present comparison of cache and backend server response time in Table 7.48. In every machine, TC2's response time is 29% faster than TC0's. The condition holds upto 6-7 threads per machine. TC1's response time is 85% faster than TC3's. The response time for cache is 5 to 6 times higher than backends. TC6's response time is 23% and 18% slower than TC8's and TC10's. TC7's response time is 40% and 34% slower than TC9's and TC11's. TC12's response time is almost the same to TC14's. TC13's response time is 83% faster than TC15's as the server takes some extra time to translate a HTTPS request to HTTP request. TC18's response time is 4% faster than TC20's and TC22's as cache hit and miss in without load condition. TC19's response time is 29% and 36% slower than TC21's and TC23's.

In addition, we present comparison of HTTP and HTTPS requests response time in Table 7.49. Here, HTTP type test cases are up to 90% faster than HTTPS protocol type as more times needed for resolving SSL/TLS keys. Besides, TC0, TC1, TC2, TC3, TC6, TC7, TC8, TC9, TC10, and TC11 are 91%, 77%, 87%, 47%, 91%, 71%, 90%, 64%, 90%, 50% faster than TC12, TC13, TC14, TC15, TC18, TC19, TC20, TC21, TC22, and TC23 respectively. In summary, we conclude that for each test case scenario in every machine, response time for cache is lower than the backend except for the same URL requests. Varnish restricts concurrent request at a time for same URL's.

7.5 Conclusion and Future Work

The main goal of this literature is to find out the sustainability of the system at different test load volumes and tuning the software and network system by analyzing test responses for OpenStack Swift and Varnish. We set benchmarks according to the continuous test results. To improve the architecture and understanding the systems behavior, we generate several statistics by taking several snapshots.

Besides, snapshots are taken using SS, hTop, Jmeter graph, and Zabbix to monitor network status, CPU and memory usage, response time, throughput, etc. respectively. There is a plenty of room to extend the study. Our future plan is to perform load tests to large files such as video contents. We have to explore in future that how the system will behave for extensive PUT and POST requests as for these requests proxy server has extra overhead related to processing and I/O operations. Our future plan also aims to use more clients and expand the cache and backend servers and make an automated graphical User Interface (GUI) for smoothly and easily operating the load tests.

Chapter 8

Conclusion and Future Work

Existing studies on cloud storage ecosystem for multimedia services are yet to provide an efficient and secured solution. To address this gap, in this study, we propose a novel cloud storage ecosystem for efficient and secured multimedia services. Besides, we design several frameworks and perform experimental evaluation of our proposed method. To conclude our study, in this Chapter, we delineate the sketches and future works of our study.

8.1 Conclusion

We discuss the challenge of efficient retrieval of cloud-related images, particularly in bandwidth-constrained situations in Chapter 3. It identifies a gap in the literature regarding the storage and retrieval of progressive images and proposes an orchestration methodology to address this issue. The proposed methodology includes a new image scanning and lossy compression technique, which is tested in a real setup across two different continents. The evaluation results demonstrate a significant improvement in the performance.

Afterward, Chapter 4 highlights the challenge of managing and securing large-scale data in cloud-based media file sharing services, particularly when it comes to storing and archiving CCTV footage. We propose a new methodology using OpenStack Swift to not only store media but also perform tasks such as sorting, resizing, and security measures. We implement this methodology in a real testbed and conduct experiments to evaluate its effectiveness, demonstrating substantial performance improvements.

Then, in Chapter 5, we propose a solution to the challenge of efficient searching in OpenStack Swift using machine learning features and Elasticsearch. We also introduce a secondary objective of creating a user-centered content-based image searching system that allows users to manipulate the YOLOv4 and YOLOv8 algorithms based on their preferences. We test the viability and responsiveness of our model, finding room for improvement, and suggest three future goals to make the system more robust and easy to use.

Next, Chapter 6 focuses on three areas related to multimedia data management in the cloud that affect storage sustainability: retrieval of video streaming data, middleware placement based on responsibility, and detection and deletion of orphan garbage data. We propose a new middleware for downloading time interval playable video segments and a mechanism for removing orphan garbage data from cloud storage, which we evaluate through rigorous experimentation in a real setup.

Moreover, performing load testing, identifying system bottlenecks and tuning them accordingly are not focused much in the literature for cloud systems. Therefore, we perform our study on these aspects (in Chapter 7). Here, we investigate several load testing considering diversified real scenarios and tuning system parameters based on findings of load testing. Our tuning results in substantial improvement in most cases.

8.2 Complexity Analysis

We demonstrate time and storage complexity of different cloud storage systems and data structures. We consider Consistent Hash (CH), Content Addressable Storage (with Multi-Layer Index), Compressed Snapshot, OpenStack Swift (CH with a File-Path DB), and our proposed ecosystem. Table 8.1 presents a quantitative comparison of time and storage complexity for different object storage systems using several data structures. Here N is the total number of files in the filesystem, n represents the number of files stored in a certain directory, m is the number of direct children under a certain directory, P is the resizing and transcoding time, M is the time for object and document detection, and F denotes the size of the object.

Table 8.1: A qualitative comparison of time and storage complexity for different object storage systems using several data structures. Here, N is the total number of files in the filesystem, n represents the number of files stored in a certain directory, m is the number of direct children under a certain directory, P is the resizing and transcoding time, M is the time for object and document detection, and F denotes the size of the object.

Time complexity	Consistent Hash (CH)	Content Address-able Storage	Compressed Snapshot	OpenStack Swift	**Our proposed Ecosystem**
File Access	$O(1)$	$O(1)$	$O(N)$	$O(1)$	$O(1)$
MKDIR	$O(1)$	$O(N)$	$O(1)$	$O(1)$	$O(1)$
RMDIR, MOVE	$O(n)$	$O(N)$	$O(N)$	$O(n)$	$O(n)$
LIST	$O(N)$	$O(m)$	$O(N)$	$O(mlogN)$	$O(mlogN)$
COPY	$O(N)$	$O(N)$	$O(N)$	$O(n{+}logN)$	$O(n + logN)$
Store/PUT	-	-	-	$O(n{+}logN)$	$O(n + logN) + O(P)$
Retrive/GET	-	-	-	$O(R) + O(mlogN)$	$O(1)$
Search	-	-	-	-	$O(1) + O(M)$
Deletion daemon	-	-	-	-	$O(mlogN) + O(n + logN)$
Storage complexity	Consistent Hash (CH)	Content Address-able Storage	Compressed Snapshot	OpenStack Swift	**Our proposed Ecosystem**
Store/PUT	-	-	-	$3O(F)$	$12O(F)$
Retrive/GET	-	-	-	-	**-**
Search	-	-	-	-	**-**
Deletion daemon	-	-	-	-	**-**

8.3 Future Work

There is a plenty of room to extend the study -

- At first, our future plan includes to explore the next-generation JPEG images for further image storage quality improvement. Moreover, our plan is to create a architecture for adaptable choice of bit pixels according to bandwidth.
- Alongside, we plan to explore different techniques for adapting real-time changes in network bandwidth. We plan to do so through sensing the change and then realizing impact of the change in the image retrieval and storing processes.
- Next, our plans for the future involve extensions to the study, including implementing customized encryption-decryption algorithms, in-cloud segmentation of video files, and integrating machine learning techniques for real-time behavior detection. We plan to do so to leverage new encryption-decryption algorithms in the process of enhancing system-level performances for retrieval and storing of multimedia contents.
- Besides, our future goals include integrating the system more compactly using a desktop-based application, adding an authentication token system to keep documents safe, and using the system to store live video feeds to test its viability as a video surveillance application.
- Furthermore, our future work includes exploring SSYNC for account, container, and object servers using multiple replicas, recursive deletion daemon algorithms using different hash rings, and experimenting with different server setups through large scale simulations.
- Additionally, we plan to explore in future that how the system will behave for extensive PUT and POST requests, as the proxy server has extra overhead related to processing and I/O operations of such requests. Our future plan also aims to use more clients and expand the cache and backend servers.

List of Publications

The research conducted as part of this thesis has resulted in the following publications.

Journal Publications

1. J. Noor, M. N. Shanto, J. J. Mondal, M. G. Hossain, S. Chellappan, and A. B. M. A. A. Islam. "Orchestrating Image Retrieval and Storage over A Cloud System", IEEE Transactions on Cloud Computing. 2022 Mar 28. [1]

2. J. Noor, R. Ratul and A. B. M. A. A. Islam. "Secure Processing-aware Media Storage and Archival (SPMSA)", IEEE Transactions on Dependable and Secure Computing. [Under review].

3. J. Noor, M. R. Uday, R. Ratul, J. J. Mondal, M. S. Islam, and A. B. M. A. A. Islam. "Sherlock in OSS: A Novel Approach of Content-Based Searching in Object Storage System", IEEE Transactions of Parallel and Distributed Computing. [Under review] [247]

4. J. Noor and A. B. M. A. A. Islam. "RemOrphan: Object Storage Sustainability through Removing Offline-Processed Orphan Garbage Data", IEEE Access. [Under review]

5. J. Noor, S. I. Salim, and A. B. M. A. A. Islam, "Strategizing secured image storing and efficient image retrieval through a new cloud framework", Journal of Networks and Compututer Applications, vol. 192, 2021. [12]

6. A. Quaium, N. A. Al-Nabhan, M. Rahaman, S. I. Salim, T. R. Toha, J. Noor, M. Hossain, I. Jahan,and A. B. M. A. A. Islam. "Towards associating negative experiences and recommendations reported by Hajj pilgrims in a mass-scale survey." Heliyon (2023). [248]

7. S. I. Selim, N. A. Rahman, J. Noor, and A. B. M. A. A. Islam, "Human-Survey Interaction (HSI): A Study on Integrity of Human Data Collectors in a Mass-Scale Hajj Pilgrimage Survey", IEEE

Access, 2021. [249]

8. T. R. Toha, A. S. M. Rizvi, J. Noor, M. A. Adnan and A. B. M. A. Al Islam, "Towards Greening MapReduce Clusters Considering Both Computation Energy and Cooling Energy", in IEEE Transactions on Parallel and Distributed Systems, vol. 32, 1 April 2021. [250]

Conference Publications

1. J. Noor, H. I. Akbar, R. A. Sujon, and A. B. M. A. A. Islam, "Secure Processing-aware Media Storage (SPMS)", 36th IEEE – International Performance Computing and Communications Conference (IPCCC 2017), San Diego, California. [11]

2. J. Noor, M.G. Hossain, M. A. Alam, S. Chellappan, and A. B. M. A. A. Islam, "svLoad: An Automated Test-Driven Architecture for Load Testing in Cloud Systems", IEEE Global Communications Conference (IEEE GLOBECOM 2018), Abu Dhabi, UAE. [251]

3. S. Nasrin, T. I. M. F. Sahryer, A. B. M. A. Al Islam and J. Noor, "Feature and Performance Based Comparative Study on Serverless Frameworks," 2021 24th International Conference on Computer and Information Technology (ICCIT), 2021, pp. 1-6

4. M.Y. Ali, S. Ahmed, M.I. Hossain, A.B.M. Alim Al Islam, J. Noor. "Electronic Health Record's Security and Access Control Using Blockchain and IPFS". Proceedings of Seventh International Congress on Information and Communication Technology. Springer, Singapore, Lecture Notes in Networks and Systems, vol 447. 2022. [252]

5. J. J. Mondal, M. F. Islam, S. Zabeen, A. B. M. A. Al Islam, and J. Noor. "Note: Plant Leaf Disease Network (PLeaD-Net): Identifying Plant Leaf Diseases through Leveraging Limited-Resource Deep Convolutional Neural Network". In ACM SIGCAS/SIGCHI Conference on Computing and Sustainable Societies (COMPASS '22). Association for Computing Machinery, New York, NY, USA, 2022, pp. 668–673. [253]

6. S. M. S. Islam, R. A. Auntor, M. Islam, M. Y. H. Anik, A. B. M. A. A. Islam, and J. Noor. "Note: Towards Devising an Efficient VQA in the Bengali Language". In ACM SIGCAS/SIGCHI Conference on Computing and Sustainable Societies (COMPASS '22). Association for Computing Machinery, New York, NY, USA, 2022, pp. 632–637. [254]

7. A. Zishan and J. Noor, "Low-Cost, Low-Power, and Low-Compute Based ECG Monitoring

Systems: Comparative Analysis and Beyond", International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT), Nov 2022. [255]

8. T. Akhtar, A. B. M. A. Al Islam and J. Noor, "Speaker Identification through Gender Detection", International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT), Nov 2022. [256]

9. M. Mubtasim, A. B. M. A. Al Islam and J. Noor, "Classification of Respiratory Diseases and COVID-19 from Respiratory and Cough Sounds", International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT), Nov 2022. [257]

10. S. Nova, A. B. M. A. Al Islam and J. Noor, "IoT Based Parking System: Prospects, Challenges, and Beyond" , 3ICT, Nov 2022. [258]

11. M. S. Mustafa, J. Lisa, J. Noor, "Design and Implementation of Wireless IoT Device for Women's Safety", 9th NSysS 2022, Dec 2022. [259]

12. F. F. Khan, N. M. Hossain, M. N. H. Shanto, S. B. Anwar, J. Noor, "Mitigating DDoS Attacks Using a Resource Sharing Network", 9th NSysS 2022, Dec 2022. [260]

13. S. Misbah, J. Noor, "Use of Machine Learning and IoT for Monitoring and Tracking of Livestock", 2022 25th International Conference on Computer and Information Technology (ICCIT), Dec 2022. [261]

14. I. Miah, J. Noor, "Advanced Waterway Transport System Based on Internet of Things (IoT): A Novel Approach", 2022 25th International Conference on Computer and Information Technology (ICCIT), Dec 2022. [262]

15. T. M. Monsaif, O. F. Alif, S. D. Amarth T. A. Sadman, and J. Noor, 2022, December. A Novel Approach to Reduce Air Pollution Through Machine Learning Based PM2. 5 Prediction. In 2022 4th International Conference on Sustainable Technologies for Industry 4.0 (STI) (pp. 1-8). IEEE. [263]

Appendices

Appendix A

Request Analysis of Several HTTP Requests

A.1 Account Authentication

In these step, account authentication is done using the tempauth middleware [3]. After a successful authentication request, an authorized key is created. Response headers include the X-Storage-Token and X-Auth-Token key. It is stored in Memcache so that future storage requests can authenticate that token. We present sample console outputs of response headers for authentication request below.

```
root@ubuntu:/opt/swift# curl -v -H ‘X-Auth-User: myaccount:me’ -H ‘X-Auth-Key: secretpassword’ http://192.168.122.100:8080/auth/v1.0/
* Hostname was NOT found in DNS cache
* Trying 192.168.122.100...
* Connected to 192.168.122.100 (192.168.122.100) port 8080 (#0)
> GET /auth/v1.0/ HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.122.100:8080
> Accept: */*
> X-Auth-User: myaccount:me
> X-Auth-Key: secretpassword
```

```
>
< HTTP/1.1 200 OK
< X-Storage-Url: http://192.168.122.100:8080/v1/AUTH_myaccount
< X-Auth-Token: AUTH_tk9285653bb2c94cf683779e2879070b94
< Content-Type: text/html; charset=UTF-8
< X-Storage-Token: AUTH_tk9285653bb2c94cf683779e2879070b94
< Content-Length: 0
< X-Trans-Id: tx175906c4ed1a471e940ad-0056c0537e
< Date: Sun, 14 Feb 2016 10:14:22 GMT
<
* Connection #0 to host 192.168.122.100 left intact
```

A.2 Account Verification

After a successful authentication request, we get an authorization token. Then, we need to verify whether the stored token is valid or not for using in verification request. We present sample console outputs of response headers for verification request below.

root@ubuntu:/opt/swift# curl -v -H 'X-Storage-Token: AUTH_tk9285653bb2c94cf683779e2879070b94' http://127.0.0.1:8080/v1/AUTH_myaccount/

```
* Hostname was NOT found in DNS cache
* Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /v1/AUTH_myaccount/ HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 127.0.0.1:8080
> Accept: */*
> X-Storage-Token: AUTH_tk9285653bb2c94cf683779e2879070b94
>
> HTTP/1.1 204 No Content
> Content-Type: text/plain; charset=utf-8
> X-Account-Object-Count: 0
> X-Timestamp: 1455444974.49098
> X-Account-Bytes-Used: 0
> X-Account-Container-Count: 0
> X-Put-Timestamp: 1455444974.49098
> Content-Length: 0
> X-Trans-Id: txaf6e294e093544b7ad7d6-0056c053ee
> Date: Sun, 14 Feb 2016 10:16:14 GMT
>
* Connection #0 to host 127.0.0.1 left intact
```

A.3 Account Creation

We present sample console outputs of response headers for the request of creating an account below.

```
root@ubuntu:/opt/swift# curl -v -H ‘X-Storage-Token:
AUTH_tk9285653bb2c94cf683779e2879070b94’ -X GET http://192.168.122.100:8080
/v1/AUTH_myaccount
* Hostname was NOT found in DNS cache
* Trying 192.168.122.100...
* Connected to 192.168.122.100 (192.168.122.100) port 8080 (#0)
> GET /v1/AUTH_myaccount HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.122.100:8080
> Accept: */*
> X-Storage-Token:AUTH_tk9285653bb2c94cf683779e2879070b94
>
< HTTP/1.1 204 No Content
< Content-Type: text/plain; charset=utf-8
< X-Account-Object-Count: 0
< X-Timestamp: 1455445046.63422
< X-Account-Bytes-Used: 0
< X-Account-Container-Count: 0
< X-Put-Timestamp: 1455445046.63422
< Content-Length: 0
< X-Trans-Id: tx828d6c25af0443b695faf-0056c05436
< Date: Sun, 14 Feb 2016 10:17:26 GMT
<
* Connection #0 to host 192.168.122.100 left intact
```

A.4 Container Creation for Objects

After creating an account, we need to create a container. As our proposed architecture stores both object-type data and image-type data, we need to create two containers. We present sample console outputs of response headers for creating object type of container request below.

```
root@ubuntu:/opt/swift# curl -v -H 'X-Storage-Token:
AUTH_tk9285653bb2c94cf683779e2879070b94' -X PUT http://192.168.122.100:8080
/v1/AUTH_myaccount/object_container
* Hostname was NOT found in DNS cache
* Trying 192.168.122.100...
* Connected to 192.168.122.100 (192.168.122.100) port 8080 (#0)
> PUT /v1/AUTH_myaccount/image_container HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.122.100:8080
> Accept: */*
> X-Storage-Token:AUTH_tk9285653bb2c94cf683779e2879070b94
>
> HTTP/1.1 201 Created
> Content-Length: 0
> Content-Type: text/html; charset=UTF-8
> X-Trans-Id: tx5e44fe64839c499e801df-0056c05498
> Date: Sun, 14 Feb 2016 10:19:04 GMT
>
* Connection #0 to host 192.168.122.100 left intact
```

A.5 Container Creation for Images

We present sample console outputs of response headers for creating container for storing images below.

```
root@ubuntu:/opt/swift# curl -v -H ‘X-Storage-Token:
AUTH_tk9285653bb2c94cf683779e2879070b94’ -X PUT http://192.168.122.100:8080
/v1/AUTH_myaccount/image_container
* Hostname was NOT found in DNS cache
* Trying 192.168.122.100...
* Connected to 192.168.122.100 (192.168.122.100) port 8080 (#0)
> PUT /v1/AUTH_myaccount/image_container HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.122.100:8080
> Accept: */*
> X-Storage-Token:AUTH_tk9285653bb2c94cf683779e2879070b94
>
> HTTP/1.1 201 Created
> Content-Length: 0
> Content-Type: text/html; charset=UTF-8
> X-Trans-Id: tx5e44fe64839c499e801df-0056c05498
> Date: Sun, 14 Feb 2016 10:19:04 GMT
>
* Connection #0 to host 192.168.122.100 left intact
```

A.6 Container Listing of a Created Account

We present sample console outputs of response headers for container listing of a created account request below.

```
root@ubuntu:/opt/swift# curl -v -H ‘X-Storage-Token:
AUTH_tk9285653bb2c94cf683779e2879070b94’ http://127.0.0.1:8080/v1
/AUTH_myaccount/
* Hostname was NOT found in DNS cache
* Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /v1/AUTH_myaccount/ HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 127.0.0.1:8080
> Accept: */*
> X-Storage-Token: AUTH_tk9285653bb2c94cf683779e2879070b94
>
< HTTP/1.1 200 OK
< Content-Length: 70
< X-Account-Object-Count: 51
< X-Account-Storage-Policy-Policy-0-Bytes-Used: 110786405
< X-Account-Storage-Policy-Policy-0-Container-Count: 6
< X-Timestamp: 1431587660.93524
< X-Account-Storage-Policy-Policy-0-Object-Count: 51
< X-Account-Bytes-Used: 110786405
< X-Account-Container-Count: 6
< Content-Type: text/plain; charset=utf-8
< Accept-Ranges: bytes
< X-Trans-Id: tx7452cd2daa9b4364b221d-0056c05528
< Date: Sun, 14 Feb 2016 10:21:28 GMT
<
image_container
jnoor
mycontainer
newcontainer
object_container
```

A.7 Object Upload

We present console outputs of response headers for uploading an object request below.

root@ubuntu:/opt/swift# curl -v -H ‘Content-Type: text/plain’ -H ‘X-Storage-

Token:AUTH_tk9285653bb2c94cf683779e2879070b94' -X PUT -T MANIFEST.in http://192.168.122.100:8080/v1/AUTH_myaccount/object_container/MANIFEST.in

```
* Hostname was NOT found in DNS cache
* Trying 192.168.122.100...
* Connected to 192.168.122.100 (192.168.122.100) port 8080 (#0)
> PUT /v1/AUTH_myaccount/object_container/MANIFEST.in HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.122.100:8080
> Accept: */*
> Content-Type: text/plain
> X-Storage-Token:AUTH_tk9285653bb2c94cf683779e2879070b94
> Content-Length: 313
> Expect: 100-continue
>
< HTTP/1.1 100 Continue
* We are completely uploaded and fine
< HTTP/1.1 201 Created
< Last-Modified: Sun, 14 Feb 2016 10:24:21 GMT
< Content-Length: 0
< Etag: 50378f00bd9d746f54ba45aabe8da742
< Content-Type: text/html; charset=UTF-8
< X-Trans-Id: tx4ac107cdded744298c2ba-0056c055d4
< Date: Sun, 14 Feb 2016 10:24:20 GMT
<
* Connection #0 to host 192.168.122.100 left intact
```

A.8 Object List of First Container

We present console outputs of response headers for object listing of a container request below.

root@ubuntu:/opt/swift# curl -v -H 'X-Storage-Token: AUTH_tk9285653bb2c94cf683779e2879070b94' -X GET http://192.168.122.100:8080

```
/v1/AUTH_myaccount/object_container
* Hostname was NOT found in DNS cache
* Trying 192.168.122.100...
* Connected to 192.168.122.100 (192.168.122.100) port 8080 (#0)
> GET /v1/AUTH_myaccount/object_container HTTP/1.1
>¿ User-Agent: curl/7.35.0
> Host: 192.168.122.100:8080
> Accept: */*
> X-Storage-Token:AUTH_tk9285653bb2c94cf683779e2879070b94
>
< HTTP/1.1 200 OK
< Content-Length: 12
< X-Container-Object-Count: 1
< Accept-Ranges: bytes
< X-Storage-Policy: Policy-0
< X-Container-Bytes-Used: 313
< X-Timestamp: 1455445169.66486
< Content-Type: text/plain; charset=utf-8
< X-Trans-Id: tx3d9f5b532b7645349fdb9-0056c05614
< Date: Sun, 14 Feb 2016 10:25:24 GMT
< MANIFEST.in
* Connection #0 to host 192.168.122.100 left intact
```

A.9 Object Download

We present sample console outputs of response headers for downloading an object request below.

```
[root@Centos7SwiftAllInOne199 home]# curl -v -o down_MANIFEST.in
-H ‘X-Auth-Token: AUTH_tk9285653bb2c94cf683779e2879070b94’
http://192.168.122.100:8080/v1/AUTH_myaccount/object_container/MANIFEST.in
* Hostname was NOT found in DNS cache
* Trying 192.168.122.100...
```

```
% Total % Received % Xferd Average Speed Time Time Time Current Dload Upload Total Spend
Left Speed
0 0 0 0 0 0 0 0 –:–:– –:–:– –:–:–
0* Connected to 192.168.122.100 (192.168.122.100) port 8080 (#0)
> GET /v1/AUTH_myaccount/object_container/MANIFEST.in HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.122.100:8080
> Accept: */*
> X-Auth-Token: AUTH_tk9285653bb2c94cf683779e2879070b94
>
< HTTP/1.1 200 OK
> Content-Length: 313
> Accept-Ranges: bytes
> Last-Modified: Sun, 14 Feb 2016 10:24:21 GMT
> Etag: 50378f00bd9d746f54ba45aabe8da742
> X-Timestamp: 1455445460.62950
> Content-Type: text/plain
> X-Trans-Id: txcef13b758df7492595735-0056c056ab
> Date: Sun, 14 Feb 2016 10:27:55 GMT
>
[data not shown]
100 313 100 313 0 0 11820 0 –:–:– –:–:– –:–:– 12520
* Connection #0 to host 192.168.122.100 left intact
```

root@ubuntu:/opt/swift# ls

AUTHORS CONTRIBUTING.md MANIFEST.in babel.cfg build **down_MANIFEST.in** examples requirements.txt setup.py swift.egg-info test-requirements.txt CHANGELOG LICENSE README.md bin doc etc pbr-0.10.8-py2.7.egg setup.cfg swift test tox.ini

A.10 Image Upload

We present console outputs of response headers for uploading an image request below.

```
root@ubuntu:/home/jannatun# curl -v -H 'Content-Type: text/plain' -H 'X-Storage-Token:AUTH_tk9285653bb2c94cf683779e2879070b94' -H 'X_IBUCK_ENABLE:1' -X PUT -T test-image.jpg http://192.168.122.100:8080/v1/AUTH_myaccount/image_container/test-image.jpg
* Hostname was NOT found in DNS cache
* Trying 192.168.122.100...
* Connected to 192.168.122.100 (192.168.122.100) port 8080 (#0)
> PUT /v1/AUTH_myaccount/image_container/test-image.jpg HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.122.100:8080
> Accept: */*
> Content-Type: text/plain
> X-Storage-Token:AUTH_tk9285653bb2c94cf683779e2879070b94
> X_IBUCK_ENABLE:1
> Content-Length: 24898
> Expect: 100-continue
>
< HTTP/1.1 100 Continue
* We are completely uploaded and fine
< HTTP/1.1 201 Created
< Content-Type: text/html; charset=UTF-8
< Content-Length: 0
< X-Trans-Id: tx0e9e8d17f2b341cfba986-0056c05c12
< Date: Sun, 14 Feb 2016 10:51:00 GMT
< * Connection #0 to host 192.168.122.100 left intact
```

A.11 Image List for Second Container

We present sample console outputs of response headers for image listing of a container request below.

```
root@ubuntu:/home/jannatun# curl -v -H ‘X-Storage-Token:
AUTH_tk9285653bb2c94cf683779e2879070b94’ -X GET http://192.168.122.100:8080
/v1/AUTH_myaccount/image_container
* Hostname was NOT found in DNS cache
* Trying 192.168.122.100...
* Connected to 192.168.122.100 (192.168.122.100) port 8080 (#0)
> GET /v1/AUTH_myaccount/image_container HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.122.100:8080
> Accept: */*
> X-Storage-Token:AUTH_tk9285653bb2c94cf683779e2879070b94
>
< HTTP/1.1 200 OK
< Content-Length: 178
< X-Container-Object-Count: 8
< Accept-Ranges: bytes
< X-Storage-Policy: Policy-0
< X-Container-Bytes-Used: 82790
< X-Timestamp: 1455445144.80031
< Content-Type: text/plain; charset=utf-8
< X-Trans-Id: txa3c26f65b3154af8a94a2-0056c05c45
< Date: Sun, 14 Feb 2016 10:51:49 GMT
<
300test-image.jpg
600test-image.jpg
p300test-image.jpg
p600test-image.jpg
test-image.jpg
```

```
ptest-image.jpg
pthumbtest-image.jpg
thumbtest-image.jpg
* Connection #0 to host 192.168.122.100 left intact
```

A.12 Another Image Upload

We present sample console outputs of response headers for uploading an image request below.

```
root@ubuntu:/home/jannatun# curl -v -H ‘Content-Type: text/plain’ -H ‘X-Storage-
Token:AUTH_tk9285653bb2c94cf683779e2879070b94’ -H ’X_IBUCK_ENABLE:1’
-X PUT -T pray.jpg http://192.168.122.100:8080/v1
/AUTH_myaccount/image_container/pray.jpg
* Hostname was NOT found in DNS cache
* Trying 192.168.122.100...
* Connected to 192.168.122.100 (192.168.122.100) port 8080 (#0)
> PUT /v1/AUTH_myaccount/image_container/pray.jpg HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.122.100:8080
> Accept: */*
> Content-Type: text/plain
> X-Storage-Token:AUTH_tk9285653bb2c94cf683779e2879070b94
> X_IBUCK_ENABLE:1
> Content-Length: 18603
> Expect: 100-continue
>
< HTTP/1.1 100 Continue
* We are completely uploaded and fine
< HTTP/1.1 201 Created
< Content-Type: text/html; charset=UTF-8
< Content-Length: 0
< X-Trans-Id: txfddba8cc6b9d4398ad234-0056c05ca5
< Date: Sun, 14 Feb 2016 10:53:27 GMT
<
* Connection #0 to host 192.168.122.100 left intact
```

A.13 Image List for Second Container

We present sample console outputs of response headers for image listing of a container request below.

```
root@ubuntu:/home/jannatun# curl -v -H 'X-Storage-Token:
AUTH_tk9285653bb2c94cf683779e2879070b94' -X GET http://192.168.122.100:8080
/v1/AUTH_myaccount/image_container
* Hostname was NOT found in DNS cache
* Trying 192.168.122.100...
* Connected to 192.168.122.100 (192.168.122.100) port 8080 (#0)
> GET /v1/AUTH_myaccount/image_container HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.122.100:8080
> Accept: */*
> X-Storage-Token:AUTH_tk9285653bb2c94cf683779e2879070b94
>
< HTTP/1.1 200 OK
< Content-Length: 276
< X-Container-Object-Count: 16
< Accept-Ranges: bytes
< X-Storage-Policy: Policy-0
< X-Container-Bytes-Used: 187367
< X-Timestamp: 1455445144.80031
< Content-Type: text/plain; charset=utf-8
< X-Trans-Id: tx5e9dac6ea8f44bc6b377f-0056c05cd8
< Date: Sun, 14 Feb 2016 10:54:16 GMT
<
300test-image.jpg
```

```
300pray.jpg
600test-image.jpg
600pray.jpg
p300test-image.jpg
p300pray.jpg
p600test-image.jpg
p600pray.jpg
test-image.jpg
ptest-image.jpg
ppray.jpg
pray.jpg
pthumbtest-image.jpg
pthumbpray.jpg
thumbtest-image.jpg
thumbpray.jpg
* Connection #0 to host 192.168.122.100 left intact
```

Appendix B

Documents for Keyword Extraction

B.1 Document A

This document has been collected from the SemEval2017 Dataset [199]. The document is as followed:

"Deep learning (also known as deep structured learning) is part of a broader family of machine learning methods based on artificial neural networks with representation learning. Learning can be supervised, semi-supervised or unsupervised. Deep learning architectures such as deep neural networks, deep belief networks, recurrent neural networks and convolutional neural networks have been applied to fields including computer vision, machine vision, speech recognition, natural language processing, audio recognition, social network filtering, machine translation, bioinformatics, drug design, medical image analysis, material inspection and board game programs, where they have produced results comparable to and in some cases surpassing human expert performance. Artificial neural networks (ANNs) were inspired by information processing and distributed communication nodes in biological systems. ANNs have various differences from biological brains. Specifically, neural networks tend to be static and sym bolic, while the biological brain of most living organisms is dynamic (plastic) and analog. The adjective "deep" in deep learning comes from the use of multiple layers in the network. Early work showed that a linear perceptron cannot be a universal classifier, and then that a network with a nonpolynomial activation function with one hidden layer of unbounded width can on the other hand so be. Deep learning is a modern variation which is concerned with an unbounded number of layers of bounded size, which per mits practical application and optimized implementation, while retaining theoretical

universality under mild conditions. In deep learning the layers are also permitted to be heterogeneous and to deviate widely from biologically informed connectionist model s, for the sake of efficiency, trainability and understandability, whence the "structured" part."

B.2 Document B

This document has been collected from Aarts et al. [264] The document is as followed:

"Complex Langevin (CL) dynamics [1,2] provides an approach to circumvent the sign problem in numerical simulations of lattice field theories with a complex Boltzmann weight, since it does not rely on importance sampling. In recent years a number of stimulating results has been obtained in the context of nonzero chemical potential, in both lower and four-dimensional field theories with a severe sign problem in the thermodynamic limit [3–8] (for two recent reviews, see e.g. Refs. [9,10]). However, as has been known since shortly after its inception, correct results are not guaranteed [11–16]. This calls for an improved understanding, relying on the combination of analytical and numerical insight. In the recent past, the important role played by the properties of the real and positive probability distribution in the complexified configuration space, which is effectively sampled during the Langevin process, has been clarified [17,18]. An important conclusion was that this distribution should be sufficiently localised in order for CL to yield valid results. Importantly, this insight has recently also led to promising results in nonabelian gauge theories, with the implementation of SL(N,C) gauge cooling [8,10]."

Bibliography

[1] J. Noor, M. N. H. Shanto, J. J. Mondal, M. G. Hossain, S. Chellappan, and A. A. Al Islam, "Orchestrating image retrieval and storage over a cloud system," *IEEE Transactions on Cloud Computing*, 2022.

[2] P. Savvaidis, "From desktop gis to web-based cloud gis: The globalization of geospatial data management." [Online]. Available: `https://www.researchgate.net/The-architecture-of-Cloud-Computing-Image-from_fig3_237009975`. Accessed: Mar. 28, 2023.

[3] J. Arnold, *OpenStack Swift: Using, administering, and developing for swift object storage.* " O'Reilly Media, Inc.", 2014.

[4] "Openstack foundation, 'welcome to swift's documentation!." `http://docs.openstack.org/developer/swift/`. Accessed: Mar. 28, 2023.

[5] J. Danjou, "Openstack swift eventual consistency analysis & bottlenecks." [Online]. Available: `https://julien.danjou.info/openstack-swift-consistency-analysis/`. Accessed: Mar. 28, 2023.

[6] K. Chen, C. Wu, Y. Chang, and C. Lei, "A crowdsourceable qoe evaluation framework for multimedia content." [Online]. Available: `https://www.slideshare.net/mmnet/a-crowdsourceable-qoe-evaluation-framework-for-multimedia-content`. Accessed: Mar. 28, 2023.

[7] L. Miller, "What is image quality assessment?." [Online]. Available: `https://slideplayer.com/slide/7688464/`. Accessed: Mar. 28, 2023.

[8] "Httparchive." [Online]. Available: http://httparchive.org/. Accessed: Mar. 28, 2023.

[9] "Unraveling the JPEG." https://parametric.press/issue-01/unraveling-the-jpeg/. Accessed: Mar. 28, 2023.

[10] R. Coyne, "Visualising 2d discrete cosine transforms." [Online]. Available: https://richardcoyne.com/2020/08/22/visualising-2d-discrete-cosine-transforms/. Accessed: Mar. 1, 2023.

[11] J. Noor, H. I. Akbar, R. A. Sujon, and A. A. Al Islam, "Secure processing-aware media storage (spms)," in *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pp. 1–8, IEEE, 2017.

[12] J. Noor, S. I. Salim, and A. A. Al Islam, "Strategizing secured image storing and efficient image retrieval through a new cloud framework," *Journal of Network and Computer Applications*, vol. 192, p. 103167, 2021.

[13] L. Tan and J. Jiang, "Chapter 13 - image processing basics," in *Digital Signal Processing (Third Edition)* (L. Tan and J. Jiang, eds.), pp. 649–726, Academic Press, third edition ed., 2019.

[14] Libjpeg-turbo, "libjpeg-turbo default scan script." [Online]. Available: https://github.com/libjpeg-turbo/libjpeg-turbo/blob/master/wizard.txt. Accessed: Jun. 04, 2023.

[15] D.Vatolin, A.Moskvin, O.Petrov, and N.Trunichkin, "Msu video quality measurement tools, [online]. available:." https://www.compression.ru/video/quality_measure/. Accessed: Mar. 28, 2023.

[16] "Cctv vs. ip cameras: Which is best suited for your business?." https://www.taylored.com/blog/cctv-vs-ip-cameras-which-is-best-suited-for-your-business/. Accessed: Mar. 28, 2023.

[17] S. Idrees, S. Nazir, S. Tahir, and M. S. Khan, "Cloud ecosystem-prevalent threats and countermeasures," in *Handbook of Research on Cybersecurity Issues and Challenges for Business and FinTech Applications*, pp. 146–173, IGI Global, 2023.

[18] A. Mali, A. Ororbia, D. Kifer, and L. Giles, "Neural jpeg: End-to-end image compression leveraging a standard jpeg encoder-decoder," *arXiv preprint arXiv:2201.11795*, 2022.

[19] P. Mullan, C. Riess, and F. Freiling, “Forensic source identification using jpeg image headers: The case of smartphones,” *Digital Investigation*, vol. 28, pp. S68–S76, 2019.

[20] A. P. Byju, B. Demir, and L. Bruzzone, “A progressive content-based image retrieval in jpeg 2000 compressed remote sensing archives,” *IEEE Transactions on Geoscience and Remote Sensing*, pp. 1–13, Feb. 2020.

[21] C. Harrison, A. K. Dey, and S. Hudson, “Evaluation of progressive image loading schemes,” in *SIGCHI Conference on Human Factors in Computing Systems (CHI’10)*, (Atlanta, Georgia, USA), pp. 1549–1552, Apr. 10-15, 2010.

[22] W.-B. Kim and I.-Y. Lee, “Secure and efficient storage of video data in a cctv environment.,” *TIIS*, vol. 13, no. 6, pp. 3238–3257, 2019.

[23] J. Kim, N. Park, G. Kim, and S. Jin, “Cctv video processing metadata security scheme using character order preserving-transformation in the emerging multimedia,” *Electronics*, vol. 8, no. 4, p. 412, 2019.

[24] R. Ashraf, M. Ahmed, U. Ahmad, M. A. Habib, S. Jabbar, and K. Naseer, “Mdcbir-mf: multimedia data for content-based image retrieval by using multiple features,” *Multimedia tools and applications*, vol. 79, no. 13, pp. 8553–8579, 2020.

[25] K. T. Ahmed, S. Ummesafi, and A. Iqbal, “Content based image retrieval using image features information fusion,” *Information Fusion*, vol. 51, pp. 76–99, 2019.

[26] S. Deniziak and T. Michno, “New content based image retrieval database structure using query by approximate shapes,” in *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 613–621, IEEE, 2017.

[27] A. Nazir, R. Ashraf, T. Hamdani, and N. Ali, “Content based image retrieval system by using hsv color histogram, discrete wavelet transform and edge histogram descriptor,” in *2018 international conference on computing, mathematics and engineering technologies (iCoMET)*, pp. 1–6, IEEE, 2018.

[28] L. Goddard and D. Seeman, “Negotiating sustainability: Building digital humanities projects that last,” in *Doing More Digital Humanities*, pp. 38–57, Routledge, 2019.

[29] C. J. Corbett, "How sustainable is big data?," *Production and Operations Management*, vol. 27, no. 9, pp. 1685–1695, 2018.

[30] N. Al-Nabhan, S. Alenazi, S. Alquwaifili, S. Alzamzami, L. Altwayan, N. Alaloula, R. Alowaini, and A. A. Al Islam, "An intelligent iot approach for analyzing and managing crowds," *IEEE Access*, vol. 9, pp. 104874–104886, 2021.

[31] M. Yamin, "Managing crowds with technology: cases of hajj and kumbh mela," *International journal of information technology*, vol. 11, no. 2, pp. 229–237, 2019.

[32] B. Hou, S. Yang, F. A. Kuipers, L. Jiao, and X. Fu, "Eavs: Edge-assisted adaptive video streaming with fine-grained serverless pipelines," in *INFOCOM 2023-IEEE International Conference on Computer Communications*, IEEE, 2023.

[33] D. Kobayashi, K. Nakamura, M. Kitahara, T. Osawa, Y. Omori, T. Onishi, and H. Iwasaki, "A low-latency 4k hevc multi-channel encoding system with content-aware bitrate control for live streaming," *IEICE TRANSACTIONS on Information and Systems*, vol. 106, no. 1, pp. 46–57, 2023.

[34] G. Zhou, Z. Luo, M. Hu, and D. Wu, "Presr: Neural-enhanced adaptive streaming of vbr-encoded videos with selective prefetching," *IEEE Transactions on Broadcasting*, 2022.

[35] A. Barakabitze and A. Hines, "Multimedia streaming services over the internet," 2023.

[36] T. Louati, H. Abbes, C. Cérin, and M. Jemni, "Gc-cr: a decentralized garbage collector component for checkpointing in clouds," in *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 97–104, IEEE, 2017.

[37] S. J. Ramson and D. J. Moni, "Wireless sensor networks based smart bin," *Computers & Electrical Engineering*, vol. 64, pp. 337–353, 2017.

[38] J. Joshi, J. Reddy, P. Reddy, A. Agarwal, R. Agarwal, A. Bagga, and A. Bhargava, "Cloud computing based smart garbage monitoring system," in *2016 3rd International Conference on Electronic Design (ICED)*, pp. 70–75, IEEE, 2016.

[39] K. M. Ramokapane, A. Rashid, and J. M. Such, "Assured deletion in the cloud: requirements, challenges and future directions," in *Proceedings of the 2016 ACM on Cloud Computing Security*

Workshop, pp. 97–108, 2016.

[40] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, and Y. J. Song, "Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Nov. 2-4, 2016.

[41] M. D. Syer, W. Shang, Z. M. Jiang, and A. E. Hassan, "Continuous validation of performance test workloads.," *Automated Software Engineering*, vol. 24, pp. 189–231, Mar. 2017.

[42] R. Bhatia and A. Ganpati, "In depth analysis of web performance testing tools," *IRACST – Engineering Science and Technology: An International Journal (ESTIJ), ISSN: 2250-3498*, vol. 6, Sep.-Oct. 2016.

[43] Sourceforge, "Libjpeg." [Online]. Available: `http://libjpeg.sourceforge.net/`. Accessed: Apr. 20, 2023.

[44] S. Gole, "Why you need cloud storage in your life." [Online]. Available: `https://cyberchimps.com/need-cloud-storage-life/`. Accessed: Mar. 28, 2023.

[45] H. J. Kim, D. H. Lee, J. M. Lee, K. H. Lee, W. Lyu, and S. G. Choi, "The qoe evaluation method through the qos-qoe correlation model," in *2008 Fourth International Conference on Networked Computing and Advanced Information Management*, vol. 2, pp. 719–725, IEEE, 2008.

[46] Y. Wang, W. Zhou, and P. Zhang, "Implementation and Demonstration of QoE Measurement Platform," *Springer International Publishing, Part of the series Springer Briefs in Electrical and Computer Engineering*, pp. 45–57, Aug. 2016.

[47] F. Kuipers, R. Kooij, D. D. Vleeschauwer, and K. Brunnström, "Techniques for measuring quality of experience," in *International Conference on Wired/Wireless Internet Communications*, pp. 216–227, Springer, 2010.

[48] P. ITU-T RECOMMENDATION, "Subjective video quality assessment methods for multimedia applications," *Networks*, 1999.

[49] Q. Huynh-Thu and M. Ghanbari, "Scope of validity of psnr in image/video quality assessment," *Electronics letters*, vol. 44, no. 13, pp. 800–801, 2008.

[50] Z. Wang and Q. Li, "Video quality assessment using a statistical model of human visual speed perception," *JOSA A*, vol. 24, no. 12, pp. B61–B69, 2007.

[51] Z. Wang and A. C. Bovik, "Mean squared error: Love it or leave it? a new look at signal fidelity measures," *IEEE signal processing magazine*, vol. 26, no. 1, pp. 98–117, 2009.

[52] M. H. Pinson and S. Wolf, "A new standardized method for objectively measuring video quality," *IEEE Transactions on broadcasting*, vol. 50, no. 3, pp. 312–322, 2004.

[53] J. Noor and A. A. Al Islam, "ibuck: Reliable and secured image processing middleware for openstack swift," in *2017 International Conference on Networking, Systems and Security (NSysS)*, pp. 144–149, IEEE, 2017.

[54] "Understanding the most popular image file types and formats." [Online]. Available: `https://www.embedded-vision.com/sites/default/files/apress/computervisionmetrics/chapter2/9781430259299_Ch02.pdf`Accessed: May 15, 2023.

[55] "JPEG - JPEG privacy & security abstract and executive summary." `https://jpeg.org/items/20150910_privacy_security_summary.html`. Accessed: Mar. 28, 2023.

[56] J. Miano, *Compressed Image File Formats.* Addison Wesley Longman, Inc. ISBN 0-201-60443-4, 1999.

[57] "Internet speeds by country 2021." `https://worldpopulationreview.com/country-rankings/internet-speeds-by-country`. Accessed: Mar. 28, 2023.

[58] "How COVID-19 is affecting internet performance." `https://www.fastly.com/blog/how-covid-19-is-affecting-internet-performance`. Accessed: Mar. 28, 2023.

[59] J. Noor, S. I. Salim, and A. B. M. A. A. Islam, "Strategizing secured image storing and efficient image retrieval through a new cloud framework," *J. Netw. Comput. Appl.*, vol. 192, no. 103167, p. 103167, 2021.

[60] F. Lin, H. Ngo, and C. Dow, "A cloud-based face video retrieval system with deep learning," *The Journal of Supercomputing*, Jan. 2020.

[61] H.-H. Hsu, T. K. Shih, L. H. Lin, R.-C. Chang, and H.-F. Li, "Adaptive image transmission by strategic decomposition," in *18th International Conference on Advanced Information Networking and Applications, 2004. AINA 2004.*, vol. 1, pp. 525–530, IEEE, 2004.

[62] C.-C. Chang, H.-C. Hsia, and T.-S. Chen, "A progressive image transmission scheme based on block truncation coding," in *International Conference Human Society@ Internet*, pp. 383–397, Springer, 2001.

[63] S. Amdahl, "Methods for managing progressive image delivery and devices thereof," Jan. 15 2019. US Patent App. 14/955,693.

[64] S. Stefanov, "Book of speed." [Online]. Available: `http://www.bookofspeed.com/chapter5.html`. Accessed: Mar. 28, 2023.

[65] S. Nazir, O. A. Alzubi, M. Kaleem, and H. Hamdoun, "Image subset communication for resource-constrained applications in wireless sensor networks," *Turkish Journal of Electrical Engineering and Computer Sciences*, 09 2020.

[66] A. Mammeri, A. Khoumsi, D. Ziou, and B. Hadjou, "Energy-efficient transmission scheme of jpeg images over visual sensor networks," in *2008 33rd IEEE Conference on Local Computer Networks (LCN)*, pp. 639–647, 2008.

[67] R. Steinmetz and K. Nahrstedt, *Multimedia: Computing Communications & Applications*. Prentice Hall PTR, 1995.

[68] M. Hasan, K. M. Nur, and H. B. Shakur, "An improved jpeg image compression technique based on selective quantization," *International Journal of Computer Applications (IJCA)*, 2012.

[69] J. A.-A. Mazen Abuzaher, "Jpeg based compression algorithm," *International Journal of Engineering and Applied Sciences (IJEAS)*, 2017.

[70] A. A. Hussain, G. K. AL-Khafaji, and M. M. Siddeq, "Developed jpeg algorithm applied in image compression," *International Scientific Conference of Al-Ayen University (ISCAU)*, 2020.

[71] G. Seelmann, "Improved redundancy reduction for jpeg files," in *Picture Coding Symposium*, 2007.

[72] E. Yan, K. Zhang, X. Wang, K. Strauss, and L. Ceze, "Customizing progressive jpeg for efficient image storage," in *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, (USA), USENIX Association, 2017.

[73] G. K. Wallace, "The jpeg still picture compression standard," *IEEE Transactions on Consumer Electronics*, 1991.

[74] S. Kunwar, "Image compression algorithm and jpeg standard," *International Journal of Scientific and Research Publications*, 2017.

[75] K. R. Spring, T. J. Fellers, and M. W. Davidson, "Human vision and color perception." [Online]. Available: `https://www.olympus-lifescience.com/en/microscope-resource/primer/lightandcolor/humanvisionintro/`. Accessed: Feb. 17, 2023.

[76] S. Rafatirad and A. Majumder, "Sensitivity to color variations." [Online]. Available: `https://www.ics.uci.edu/~majumder/vispercep/paper08/spatialvision.pdf`. Accessed: Feb. 17, 2023.

[77] Y. Khalid, "Understanding and decoding a jpeg image using python." [Online]. Available: `https://yasoob.me/posts/understanding-and-writing-jpeg-decoder-in-python/`. Accessed: Feb. 17, 2023.

[78] M. S. AL-Ani and F. H. Awad, "The jpeg image compression algorithm," *International Journal of Advances in Engineering & Technology*, 2013.

[79] V. Sharma, "Qutub complex monuments' images dataset." [Online]. Available: `https://www.kaggle.com/varunsharmaml/qutub-complex-monuments-images-dataset`. Accessed: Feb. 16, 2023.

[80] M. Oltean, "Fruits 360." [Online]. Available: `https://www.kaggle.com/moltean/fruits`. Accessed: Feb. 16, 2023.

[81] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pp. 740–755, Springer, 2014.

[82] "Structural similarity index — skimage v0.19.0.dev0 docs." `https://scikit-image.org/docs/dev/auto_examples/transform/plot_ssim.html`. Accessed: Mar. 28, 2023.

[83] "Chrome DevTools." `https://developer.chrome.com/docs/devtools/`. Accessed: Mar. 28, 2023.

[84] M. Abu-Zaher and J. Al-Azzeh, "Jpeg based compression algorithm," *International Journal of Engineering and Applied Sciences*, vol. 4, p. 257481, 2017.

[85] A. Louie and A. M. K. Cheng, "Work-in-progress: Designing a server-side progressive JPEG encoder for real-time applications," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, IEEE, 2020.

[86] Y. Iqbal and O.-J. Kwon, "Improved JPEG coding by filtering 8 $\times$ 8 DCT blocks," *J. Imaging*, vol. 7, no. 7, p. 117, 2021.

[87] A. Mali, A. Ororbia, D. Kifer, and L. Giles, "Neural JPEG: End-to-end image compression leveraging a standard JPEG encoder-decoder," *IEEE Signal Processing Society SigPort*, 2022.

[88] J. Lee, S. Jeon, K. P. Choi, Y. Park, and C. Kim, "DPICT: deep progressive image compression using trit-planes," *CoRR*, vol. abs/2112.06334, 2021.

[89] C. Cai, L. Chen, X. Zhang, G. Lu, and Z. Gao, "A novel deep progressive image compression framework," in *2019 Picture Coding Symposium (PCS)*, IEEE, 2019.

[90] Y. Lu, Y. Zhu, Y. Yang, A. Said, and T. S. Cohen, "Progressive neural image compression with nested quantization and latent ordering," in *2021 IEEE International Conference on Image Processing (ICIP)*, IEEE, 2021.

[91] N. Abdollahi, K. Shahtalebi, and M. F. Sabahi, "High compression rate, based on the RLS adaptive algorithm in progressive image transmission," *Signal Image Video Process.*, vol. 15, no. 4, pp. 835–842, 2021.

[92] R. Tyleček and R. Šára, "Spatial pattern templates for recognition of objects with regular structure," in *Pattern Recognition* (J. Weickert, M. Hein, and B. Schiele, eds.), (Berlin, Heidelberg), pp. 364–374, Springer Berlin Heidelberg, 2013.

[93] F. Korč and W. Förstner, "etrims image database for interpreting images of man-made scenes," Dept. of Photogrammetry, University of Bonn, 05 2009.

[94] M. J. Huiskes and M. S. Lew, "The mir flickr retrieval evaluation," in *Proceedings of the 1st ACM International Conference on Multimedia Information Retrieval*, MIR '08, (New York, NY, USA), p. 39–43, Association for Computing Machinery, 2008.

[95] J. Ascenso and P. Akayzi, "JPEG AI image coding common test conditions," in *Proceedings of the ISO/IEC JTC1/SC29/WG1 N84035, 84th Meeting*, 2019.

[96] R. W. Franzen, "Kodak lossless true color image suite." `http://r0k.us/graphics/kodak`. Accessed: Mar. 28, 2023.

[97] E. Agustsson and R. Timofte, "NTIRE 2017 challenge on single image Super-Resolution: Dataset and study," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, IEEE, 2017.

[98] T. Xue, B. Chen, J. Wu, D. Wei, and W. T. Freeman, "Video enhancement with task-oriented flow," *Int. J. Comput. Vis.*, vol. 127, no. 8, pp. 1106–1125, 2019.

[99] "BigEarth - accurate and scalable processing of big data in earth observation." `https://bigearth.eu/datasets.html`. Accessed: Mar. 28, 2023.

[100] M. Rabbani, "Jpeg2000: Image compression fundamentals, standards and practice," *Journal of Electronic Imaging*, vol. 11, no. 2, 2002.

[101] F. Bellard, "BPG image format." `http://bellard.org/bpg/`. Accessed: Mar. 28, 2023.

[102] J. Balle, D. Minnen, S. Singh, S. J. Hwang, and N. Johnston, "Variational image compression with a scale hyperprior," in *International Conference on Learning Representations*, 2018.

[103] Google, "WebP: Compression techniques." `http://developers.google.com/speed/webp/docs/compression`. Accessed: Mar. 28, 2023.

[104] G. Toderici, D. Vincent, N. Johnston, S. J. Hwang, D. Minnen, J. Shor, and M. Covell, "Full resolution image compression with recurrent neural networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2017.

[105] "Cloudwards, '5 best cloud storage for video 2017'." https://www.cloudwards.net/best-cloud-storage-for-video/. Accessed: Mar. 28, 2023.

[106] "Dropbox file size limits and how to upload large files." https://help.dropbox.com/installs-integrations/sync-uploads/upload-limitations/. Accessed: Mar. 28, 2023.

[107] "Sync. [online]. available:." https://www.sync.com/. Accessed: Mar. 28, 2023.

[108] "Sugarsync. [online]. available:." https://www.sugarsync.com/. Accessed: Mar. 28, 2023.

[109] "Livedrive. [online]. available:." https://www2.livedrive.com/. Accessed: Mar. 28, 2023.

[110] "Google drive. [online]. available:." https://google.com/products/drive/. Accessed: Mar. 28, 2022.

[111] "Imemories, 'the easiest way to digitize home movies and photos,' [online]. available:." http://www.imemories.com/. Accessed: Mar. 28, 2022.

[112] "Cloudinary, 'video transcoding' [online]. available:." http://cloudinary.com/features/video_transcoding. Accessed: Mar. 28, 2022.

[113] J. Kim, D. Lee, and N. Park, "Cctv-rfid enabled multifactor authentication model for secure differential level video access control," *Multimedia Tools and Applications*, vol. 79, no. 31, pp. 23461–23481, 2020.

[114] M. U. Yaseen, M. S. Zafar, A. Anjum, and R. Hill, "High performance video processing in cloud data centres," in *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pp. 152–161, IEEE, 2016.

[115] A. Alam, M. N. Khan, J. Khan, and Y.-K. Lee, "Intellibvr-intelligent large-scale video retrieval for objects and events utilizing distributed deep-learning and semantic approaches," in *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pp. 28–35, IEEE, 2020.

[116] "Wpengine, 'what is https and ssl/tls?,' [online]. available:." https://wpengine.com/support/how-does-all-this-work-https/. Accessed: Mar. 28, 2023.

[117] S. Turner, "Transport layer security," *IEEE Internet Computing*, vol. 18, no. 6, pp. 60–63, 2014.

[118] "Video surveillance storage: How much is enough?." `https://www.seagate.com/tech-insights/how-much-video-surveillance-storage-is-enough-master-ti/`. Accessed: Mar. 28, 2023.

[119] M. A. Akbar and T. N. Azhar, "Concept of cost efficient smart cctv network for cities in developing country," in *2018 International Conference on ICT for Smart Society (ICISS)*, pp. 1–4, 2018.

[120] D. A. Rodríguez-Silva, L. Adkinson-Orellana, F. J. Gonz'lez-Castaño, I. Armiño-Franco, and D. Gonz'lez-Martínez, "Video surveillance based on cloud storage," in *2012 IEEE Fifth International Conference on Cloud Computing*, pp. 991–992, 2012.

[121] "Nest cam iq indoor — the sharper home security camera." `https://nest.com/cameras/nest-cam-iq-indoor/overview`. Accessed: Mar. 28, 2023.

[122] P. Mockapetris and K. J. Dunlap, "Development of the domain name system," in *Symposium proceedings on Communications architectures and protocols*, pp. 123–133, 1988.

[123] "Openstack swift architecture." `https://www.swiftstack.com/docs/introduction/openstack_swift.html/`. Accessed: Mar. 28, 2023.

[124] D. Vatolin, A. Moskvin, O. Petrov, and N. Trunichkin, "ringid [online]. available:." `https://www.ringid.com/`. Accessed: Mar. 28, 2022.

[125] M. Montagnuolo, A. Messina, E. K. Kolodner, D. Chen, E. Rom, K. Meth, and P. Ta-Shma, "Supporting media workflows on an advanced cloud object store platform," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pp. 384–389, 2016.

[126] Y. Wang, W.-T. Chen, H. Wu, A. Kokaram, and J. Schaeffer, "A cloud-based large-scale distributed video analysis system," in *2016 IEEE International Conference on Image Processing (ICIP)*, pp. 1499–1503, IEEE, 2016.

[127] S. Rahul *et al.*, "Cloud computing: Advantages and security challenges," *International Journal of Information and Computation Technology, ISSN 0974-2239*, vol. 3, no. 8, pp. 771–778, 2013.

[128] M. Darwich, Y. Ismail, T. Darwich, and M. A. Bayoumi, "Cost-efficient storage for on-demand video streaming on cloud," *CoRR*, vol. abs/2007.03410, 2020.

[129] E. Baik, A. Pande, Z. Zheng, and P. Mohapatra, "Vsync: Cloud based video streaming service for mobile devices," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pp. 1–9, IEEE, 2016.

[130] Z. Liu, Q. Wang, J. Huang, Y. Wu, Y. Wang, X. Jia, and H. Chen, "Cloud-based video-on-demand services for smart tv," in *2017 Seventh International Conference on Information Science and Technology (ICIST)*, pp. 81–84, IEEE, 2017.

[131] K. I. Choi, J. H. Lee, and B. C. Lee, "Cloud based video storage system with privacy protection," in *2015 17th International Conference on Advanced Communication Technology (ICACT)*, pp. 460–463, IEEE, 2015.

[132] S. Saon, H. Hashim, M. A. Ahmadon, S. Yamaguchi, *et al.*, "Cloud-based people counter," *Bulletin of Electrical Engineering and Informatics*, vol. 9, no. 1, pp. 284–291, 2020.

[133] A. Sipser, *Video ingress system for surveillance video querying.* PhD thesis, Massachusetts Institute of Technology, 2020.

[134] N. Park and N. Kang, "Mutual authentication scheme in secure internet of things technology for comfortable lifestyle," *Sensors*, vol. 16, no. 1, p. 20, 2016.

[135] M. Ali, A. Anjum, O. Rana, A. R. Zamani, D. Balouek-Thomert, and M. Parashar, "Res: Real-time video stream analytics using edge enhanced clouds," *IEEE Transactions on Cloud Computing*, 2020.

[136] H. Wang, S. Mehrotra, M. Martini, D. Wu, and Q. Zhang, "Guest editorial: Cloud-based video processing and content sharing," *IEEE Transactions on Multimedia*, vol. 18, no. 5, pp. 805–806, 2016.

[137] N. Narang, "Technical series : Handy ffmpeg commands for all video processing needs, [online]. available:." `http://www.mediaentertainmentinfo.com/2015/06/5-technical-series-handy-ffmpeg-commands-to-get-your-video-processing-done.html/`. Accessed: Mar. 28, 2023.

[138] P. Rad, M. Muppidi, S. S. Agaian, and M. Jamshidi, "Secure image processing inside cloud file sharing environment using lightweight containers," in *2015 IEEE International Conference on Imaging Systems and Techniques (IST)*, pp. 1–6, 2015.

[139] "'python cryptography toolkit (pycrypto),' [online]. available:." `https://pypi.python.org/pypi/pycrypto/`. Accessed: Mar. 28, 2023.

[140] "Understanding camera specifications." `https://clearview-communications.com/wp-content/uploads/2017/10/Understanding-the-CCTV-specifications.pdf/`. Accessed: Mar. 28, 2023.

[141] "Openstack swift architecture - data storage is changing." `https://www.swiftstack.com/docs/introduction/openstack{_}swift.html`. Accessed: Mar. 28, 2023.

[142] E. Blasch, Z. Wang, H. Ling, K. Palaniappan, G. Chen, D. Shen, A. Aved, and G. Seetharaman, "Video-based activity analysis using the l1 tracker on virat data," in *2013 IEEE Applied Imagery Pattern Recognition Workshop (AIPR)*, pp. 1–8, IEEE, 2013.

[143] S. Jaswal and M. Malhotra, "A detailed analysis of trust models in cloud environment," in *Proceedings of the Second International Conference on Data Science, E-Learning and Information Systems*, pp. 1–5, 2019.

[144] F. Duarte, "Amount of data created daily (2023)." `https://explodingtopics.com/blog/data-generated-per-day`. Accessed: Mar. 28, 2023.

[145] M. Imran and H. Hlavacs, "Searching in cloud object storage by using a metadata model," in *2013 Ninth International Conference on Semantics, Knowledge and Grids*, pp. 121–128, IEEE, 2013.

[146] O'Reilly, "What's wrong with object storage?." `https://www.networkcomputing.com/data-centers/whats-wrong-object-storage`. Accessed: Mar. 28, 2023.

[147] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[148] G. Jocher, A. Chaurasia, and J. Qiu, "Yolo by ultralytics.." `https://github.com/ultralytics/ultralytics`. Accessed: Mar. 28, 2023.

[149] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018.

[150] C. Gormley and Z. Tong, *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* " O'Reilly Media, Inc.", 2015.

[151] A. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. Miller, "High-performance metadata indexing and search in petascale data storage systems," in *Journal of Physics: Conference Series*, vol. 125, p. 012069, IOP Publishing, 2008.

[152] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman, "A metadata catalog service for data intensive applications," in *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, pp. 33–33, IEEE, 2003.

[153] M. Rostanski, K. Grochla, and A. Seman, "Evaluation of highly available and fault-tolerant middleware clustered architectures using rabbitmq," in *2014 federated conference on computer science and information systems*, pp. 879–884, IEEE, 2014.

[154] H. Ren, Z. Zheng, Y. Wu, H. Lu, Y. Yang, Y. Shan, and S.-K. Yeung, "ACNet: Approaching-and-centralizing network for zero-shot sketch-based image retrieval," *IEEE Transactions on Circuits and Systems for Video Technology*, pp. 1–1, 2023.

[155] K. Kakizaki, K. Fukuchi, and J. Sakuma, "Certified defense for content based image retrieval," in *2023 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pp. 4550–4559, 2023.

[156] Y. Wang, L. Chen, G. Wu, K. Yu, and T. Lu, "Efficient and secure content-based image retrieval with deep neural networks in the mobile cloud computing," *Computers & Security*, vol. 128, p. 103163, 2023.

[157] J. Choe, H. J. Hwang, J. B. Seo, S. M. Lee, J. Yun, M.-J. Kim, J. Jeong, Y. Lee, K. Jin, R. Park, J. Kim, H. Jeon, N. Kim, J. Yi, D. Yu, and B. Kim, "Content-based image retrieval by using deep learning for interstitial lung disease diagnosis with chest CT," *Radiology*, vol. 302, no. 1, pp. 187–197, 2022.

[158] M. M. Monowar, M. A. Hamid, A. Q. Ohi, M. O. Alassafi, and M. F. Mridha, "AutoRet: A self-supervised spatial recurrent network for content-based image retrieval," *Sensors (Basel)*, vol. 22, no. 6, p. 2188, 2022.

[159] N. Keisham and A. Neelima, "Efficient content-based image retrieval using deep search and rescue algorithm," *Soft Comput.*, vol. 26, no. 4, pp. 1597–1616, 2022.

[160] H. Ahmadvand, F. Foroutan, and M. Fathy, "Dv-dvfs: merging data variety and dvfs technique to manage the energy consumption of big data processing," *Journal of Big Data*, vol. 8, pp. 1–16, 2021.

[161] S. Xue, C. Wen, X. Zhang, and Z. Wang, "Optimization scheme of massive meteorological data storage based on openstack swift," in *2020 12th International Conference on Communication Software and Networks (ICCSN)*, pp. 302–306, 2020.

[162] S. Lima, A. Rocha, and L. Roque, "An overview of openstack architecture: a message queuing services node," *Cluster Computing*, vol. 22, no. 3, pp. 7087–7098, 2019.

[163] Evans, "Object storage essential capabilities #3 – searching, indexing and metadata." `https://www.architecting.it/blog/object-storage-critical-capabilities-3-searching-indexing-and-metadata/`. Accessed: Mar. 28, 2023.

[164] S. Gugnani, X. Lu, and D. K. Panda, "Swift-x: Accelerating openstack swift with rdma for building an efficient hpc cloud," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 238–247, IEEE, 2017.

[165] Yigal, "Openstack monitoring with elasticsearch, logstash, and kibana." `https://logz.io/blog/openstack-monitoring/`. Accessed: Mar. 28, 2023.

[166] P. Biswas, F. Patwa, and R. Sandhu, "Content level access control for openstack swift storage," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pp. 123–126, 2015.

[167] H. Wang, Y. Cai, Y. Zhang, H. Pan, W. Lv, and H. Han, "Deep learning for image retrieval: What works and what doesn't," in *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, pp. 1576–1583, IEEE, 2015.

[168] A. Raghuveer, M. Jindal, M. F. Mokbel, B. Debnath, and D. Du, "Towards efficient search on unstructured data: an intelligent-storage approach," in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pp. 951–954, 2007.

[169] S. A. Brandt, E. L. Miller, D. D. Long, and L. Xue, "Efficient metadata management in large distributed storage systems," in *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings.*, pp. 290–298, IEEE, 2003.

[170] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[171] "Search documents and images stored in object storage using opensearch, oci vision, text recognition." `https://docs.oracle.com/en/solutions/oci-opensearch-vision/index.html`. Accessed: Mar. 28, 2023.

[172] S. Anjanadevi, D. Vijayakumar, and D. K. Srinivasagan, "An efficient dynamic indexing and metadata model for storage in cloud environment," *Networking and Communication Engineering*, vol. 6, no. 3, pp. 124–129, 2014.

[173] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[174] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[175] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[176] K. Schall, K. U. Barthel, N. Hezel, and K. Jung, "Gpr1200: A benchmark for general-purpose content-based image retrieval," in *MultiMedia Modeling: 28th International Conference, MMM 2022, Phu Quoc, Vietnam, June 6–10, 2022, Proceedings, Part I*, pp. 205–216, Springer, 2022.

[177] H. Ahmadvand, M. Goudarzi, and F. Foroutan, "Gapprox: using gallup approach for approximation in big data processing," *Journal of Big Data*, vol. 6, pp. 1–24, 2019.

[178] H. Ahmadvand and F. Foroutan, "Dv-arpa: data variety aware resource provisioning for big data processing in accumulative applications," *arXiv preprint arXiv:2008.04674*, 2020.

[179] H. Ahmadvand and M. Goudarzi, "Sair: significance-aware approach to improve qor of big data processing in case of budget constraint," *The Journal of Supercomputing*, vol. 75, pp. 5760–5781, 2019.

[180] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: an astounding baseline for recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pp. 806–813, 2014.

[181] M. Koskela and J. Laaksonen, "Convolutional network features for scene recognition," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 1169–1172, 2014.

[182] N. Nikzad-Khasmakhi, M.-R. Feizi-Derakhshi, M. Asgari-Chenaghlu, M.-A. Balafar, A.-R. Feizi-Derakhshi, T. Rahkar-Farshi, M. Ramezani, Z. Jahanbakhsh-Nagadeh, E. Zafarani-Moattar, and M. Ranjbar-Khadivi, "Phraseformer: Multimodal key-phrase extraction using transformer and graph embedding," *arXiv preprint arXiv:2106.04939*, 2021.

[183] A. Xiong, D. Liu, H. Tian, Z. Liu, P. Yu, and M. Kadoch, "News keyword extraction algorithm based on semantic clustering and word graph model," *Tsinghua Sci. Technol.*, vol. 26, no. 6, pp. 886–893, 2021.

[184] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.

[185] H. Ma, Y. Liu, Y. Ren, and J. Yu, "Detection of collapsed buildings in post-earthquake remote sensing images based on the improved yolov3," *Remote Sensing*, vol. 12, no. 1, p. 44, 2020.

[186] C.-Y. Wang, H.-Y. Mark Liao, Y.-H. Wu, P.-Y. Chen, J.-W. Hsieh, and I.-H. Yeh, "CSPNet: A new backbone that can enhance learning capability of CNN," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, IEEE, 2020.

[187] A. Neubeck and L. Van Gool, "Efficient Non-Maximum suppression," in *18th International Conference on Pattern Recognition (ICPR'06)*, IEEE, 2006.

[188] Z. Zheng, P. Wang, D. Ren, W. Liu, R. Ye, Q. Hu, and W. Zuo, "Enhancing geometric factors in model learning and inference for object detection and instance segmentation," *IEEE Trans. Cybern.*, vol. 52, no. 8, pp. 8574–8586, 2022.

[189] "What is yolov8? the ultimate guide.." `https://blog.roboflow.com/whats-new-in-yolov8/`. Accessed: Mar. 28, 2022.

[190] "Yolov8_whaosoft143's blog - csdn blog." `https://blog.csdn.net/qq_29788741/article/details/128626422`. Accessed: Mar. 28, 2023.

[191] "A comprehensive review of yolo: From yolov1 to yolov8 and beyond." `https://arxiv.org/pdf/2304.00501.pdf`. Accessed: Mar. 28, 2022.

[192] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[193] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," 2013.

[194] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Ł. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016.

[195] "Elasticsearch architecture: 7 key components." `https://cloud.netapp.com/blog/cvo-blg-elasticsearch-architecture-7-key-components`. Accessed: Mar. 28, 2023.

[196] A. Iyengar *et al.*, "Enhanced storage clients," *US Appl*, no. 14/985,509, 2015.

[197] E. Bacis, S. De Capitani di Vimercati, S. Foresti, D. Guttadoro, S. Paraboschi, M. Rosa, P. Samarati, and A. Saullo, "Managing data sharing in openstack swift with over-encryption," in *Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security*, pp. 39–48, 2016.

[198] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*, pp. 740–755, Springer, 2014.

[199] I. Augenstein, M. Das, S. Riedel, L. Vikraman, and A. Mccallum, "Semeval 2017 task 10: Scienceie-extracting keyphrases and relations from scientific publications," pp. 546–555, 2017.

[200] Y. Mistry, D. Ingole, and M. Ingole, “Content based image retrieval using hybrid features and various distance metric,” *Journal of Electrical Systems and Information Technology*, vol. 5, no. 3, pp. 874–888, 2018.

[201] P. Liu, J.-M. Guo, K. Chamnongthai, and H. Prasetyo, “Fusion of color histogram and lbp-based features for texture image retrieval and classification,” *Information Sciences*, vol. 390, pp. 95–111, 2017.

[202] J. Aguilar-Armijo, C. Timmerer, and H. Hellwagner, “Space: Segment prefetching and caching at the edge for adaptive video streaming,” *IEEE Access*, vol. 11, pp. 21783–21798, 2023.

[203] L. Pu, J. Shi, X. Yuan, X. Chen, L. Jiao, T. Zhang, and J. Xu, “Ems: Erasure-coded multi-source streaming for uhd videos within cloud native 5g networks,” *IEEE Transactions on Mobile Computing*, 2023.

[204] A. Rahman, E. Hassanain, and M. S. Hossain, “Towards a secure mobile edge computing framework for hajj,” *IEEE Access*, vol. 5, pp. 11768–11781, 2017.

[205] A. Ahmad, M. A. Rahman, F. U. Rehman, A. Lbath, I. Afyouni, A. Khelil, S. O. Hussain, B. Sadiq, and M. R. Wahiddin, “A framework for crowd-sourced data collection and context-aware services in hajj and umrah,” in *2014 IEEE/ACS 11th International Conference on Computer Systems and Applications (AICCSA)*, pp. 405–412, 2014.

[206] M. Chen, “Amvsc: A framework of adaptive mobile video streaming in the cloud,” in *Globecom - Communications Software, Services and Multimedia Symposium*, 2012.

[207] X. Wang, M. Chen, T. T. Kwon, L. Yang, and V. C. Leung, “Ames-cloud: A framework of adaptive mobile video streaming and efficient social video sharing in the clouds,” *IEEE Transactions on Multimedia*, vol. 15, no. 4, pp. 811–820, 2013.

[208] E. M. Chiheb Ben Ameur and B. Cousin, “Evaluation of gateway-based shaping methods for http adaptive streaming,” in *Workshop on Quality of Experience-based Management for Future Internet Applications and Services (QoE-FI)*, IEEE ICC, 2015.

[209] M. Seufert, S. Egger, M. Slanina, T. Zinner, T. Hoßfeld, and P. Tran-Gia, “A survey on quality of experience of http adaptive streaming,” *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 469–492, 2014.

[210] C. W. Yichao Jin, Yonggang Wen and, "Optimal transcoding and caching for adaptive streaming in media cloud: An analytical approach," in *DOI 10.1109/TCSVT.2015.2402892*, IEEE Transactions on Circuits and Systems for Video Technology, 2015.

[211] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, *et al.*, "Finding a needle in haystack: Facebook's photo storage.," in *OSDI*, vol. 10, pp. 1–8, 2010.

[212] L. Xu, T. Guo, W. Dou, W. Wang, and J. Wei, "An experimental evaluation of garbage collectors on big data applications," in *The 45th International Conference on Very Large Data Bases (VLDB'19)*, 2019.

[213] R. Bruno and P. Ferreira, "A study on garbage collection algorithms for big data environments," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–35, 2018.

[214] M. Jahanshahi, K. Mostafavi, M. Kordafshari, M. Gholipour, and A. T. Haghighat, "Two new approaches for orphan detection," in *19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1 (AINA papers)*, vol. 2, pp. 461–464, IEEE, 2005.

[215] A. Sabbaghi, "A new approach to detect and eliminate orphan processes," *Journal of Mechanical Engineering and Vibration*, vol. 4, no. 3, pp. 13–19, 2013.

[216] "Ffmpeg." [Online]. Available: `https://www.ffmpeg.org/`. Accessed: Mar. 28, 2023.

[217] "Setuptools." [Online]. Available: `https://pypi.org/project/setuptools/`. Accessed: Mar. 28, 2023.

[218] A. Seema, L. Schwoebel, T. Shah, J. Morgan, and M. Reisslein, "Wvsnp-dash: Name-based segmented video streaming," *IEEE Transactions on Broadcasting*, vol. 61, no. 3, pp. 346–355, 2015.

[219] Z. Liu and Y. Wei, "Hop-by-hop adaptive video streaming in content centric network," in *2016 IEEE International Conference on Communications (ICC)*, pp. 1–7, IEEE, 2016.

[220] E. Althaus, P. Berenbrink, A. Brinkmann, and R. Steiner, "On the optimality of the greedy garbage collection strategy for ssds," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pp. 78–88, IEEE, 2022.

[221] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz, “Taurus: A holistic language runtime system for coordinating distributed managed-language applications,” *Acm SIGPLAN Notices*, vol. 51, no. 4, pp. 457–471, 2016.

[222] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng, “Lifetime-based memory management for distributed data processing systems,” *arXiv preprint arXiv:1602.01959*, 2016.

[223] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen, “Numagic: A garbage collector for big data on big numa machines,” *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 661–673, 2015.

[224] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard, “Broom: Sweeping out garbage collection from big data systems,” in *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[225] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu, “Facade: A compiler and runtime for (almost) object-bounded big data applications,” *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 675–690, 2015.

[226] “Paste deployment.” [Online]. Available: `http://pastedeploy.readthedocs.io/en/latest/#paste-filter-factory`. Accessed: Mar. 28, 2023.

[227] E. Z. T. T. Chekam and Z. Li, “On the synchronization bottleneck of openstack swift-like cloud storage systems,” in *The 35th Annual IEEE International Conference on Computer Communications*, IEEE INFOCOM, April 10-14 2016.

[228] M. Kweun, G. Kim, B. Oh, S. Jung, T. Um, and W.-Y. Lee, “Pokémem: Taming wild memory consumers in apache spark,” in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 59–69, IEEE, 2022.

[229] I. K. Kim, S. Zeng, C. Young, J. Hwang, and M. Humphrey, “icsi: A cloud garbage vm collector for addressing inactive vms with machine learning,” in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 17–28, IEEE, 2017.

[230] R. Bruno, L. P. Oliveira, and P. Ferreira, “Ng2c: pretenuring garbage collection with dynamic generations for hotspot big data applications,” in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, pp. 2–13, 2017.

[231] N. Cohen and E. Petrank, "Data structure aware garbage collector," in *Proceedings of the 2015 International Symposium on Memory Management*, pp. 28–40, 2015.

[232] G. Tene, B. Iyengar, and M. Wolf, "C4: The continuously concurrent compacting collector," in *Proceedings of the international symposium on Memory management*, pp. 79–88, 2011.

[233] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *Proceedings of the 4th international symposium on Memory management*, pp. 37–48, 2004.

[234] "Varnish cache.." [Online]. Available: `http://dx.doi.org/10.1090/S0894-0347-96-00192-0`. Accessed: Mar. 28, 2023.

[235] R. Khan and M. Amjad, "Performance testing (load) of web applications based on test case management.," *Perspectives in Science*, vol. 8, pp. 355–357, Sep. 2016.

[236] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker, "Software performance testing based on workload characterization," in *WOSP '02*, (Rome, Italy), Jul. 24-26, 2002.

[237] "Apache jmeter." [Online]. Available: `http://jmeter.apache.org/`. Accessed: Jun. 04, 2023.

[238] "Ansible." [Online]. Available: `https://www.ansible.com/`. Accessed: Jun. 04, 2023.

[239] N. Khanghahi and R. Ravanmehr, "Cloud computing performance evaluation: Issues and challenges," *International Journal on Cloud Computing: Services and Architecture (IJCCSA)*, vol. 3, Oct. 2013.

[240] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: Issues, an approach, and case study," *IEEE Transactions on Software Engineering*, vol. 26, Dec. 2000.

[241] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, J. J. L. A. Bertolino, and H. Zhu, "An orchestrated survey on automated software test case generations.," *The Journal of Systems and Software*, 2013.

[242] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," *Future of Software Engineering(FOSE'07), 0-7695-2829-5/07, IEEE.*

[243] Compuware, "Applied performance management survey," Sep.-Oct. 2006.

[244] C. U. Smith, *Performance Engineering of Software Systems.* Addison Wesley, 1990.

[245] C. Smith, *Software Performance Engineering.* Encyclopedia of Software Engineering, Wiley, 2002.

[246] "Curl.1 the man page." [Online]. Available: `https://curl.haxx.se/docs/manpage.html`. Accessed: Jun. 04, 2023.

[247] J. Noor, M. R. A. Uday, R. H. Ratul, J. J. Mondal, M. Sakif, S. Islam, and A. Islam, "Sherlock in oss: A novel approach of content-based searching in object storage system," *arXiv preprint arXiv:2303.02105*, 2023.

[248] A. Quaium, N. A. Al-Nabhan, M. Rahaman, S. I. Salim, T. R. Toha, J. Noor, M. Hossain, N. Islam, A. Mostak, M. S. Islam, *et al.*, "Towards associating negative experiences and recommendations reported by hajj pilgrims in a mass-scale survey," *Heliyon*, 2023.

[249] S. I. Salim, N. A. Al-Nabhan, M. Rahaman, N. Islam, T. R. Toha, J. Noor, A. Quaium, A. Mostak, M. Hossain, M. M. Mushfiq, *et al.*, "Human-survey interaction (hsi): A study on integrity of human data collectors in a mass-scale hajj pilgrimage survey," *IEEE Access*, vol. 9, pp. 112528–112551, 2021.

[250] T. R. Toha, A. Rizvi, J. Noor, M. A. Adnan, and A. A. Al Islam, "Towards greening mapreduce clusters considering both computation energy and cooling energy," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 931–942, 2020.

[251] J. Noor, M. G. Hossain, M. A. Alam, A. Uddin, S. Chellappan, and A. A. Al Islam, "Svload: An automated test-driven architecture for load testing in cloud systems," in *2018 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–7, IEEE, 2018.

[252] M. Y. Ali, S. Ahmed, M. I. Hossain, A. Alim Al Islam, and J. Noor, "Electronic health record's security and access control using blockchain and ipfs," in *Proceedings of Seventh International Congress on Information and Communication Technology: ICICT 2022, London, Volume 1*, pp. 493–505, Springer, 2022.

[253] J. J. Mondal, M. F. Islam, S. Zabeen, A. A. A. Islam, and J. Noor, "Note: Plant leaf disease network (plead-net): Identifying plant leaf diseases through leveraging limited-resource

deep convolutional neural network," in *ACM SIGCAS/SIGCHI Conference on Computing and Sustainable Societies (COMPASS)*, pp. 668–673, 2022.

[254] S. S. Islam, R. A. Auntor, M. Islam, M. Y. H. Anik, A. A. A. Islam, and J. Noor, "Note: Towards devising an efficient vqa in the bengali language," in *ACM SIGCAS/SIGCHI Conference on Computing and Sustainable Societies (COMPASS)*, pp. 632–637, 2022.

[255] M. A. O. Zishan, H. Shihab, S. S. Islam, M. A. Riya, G. M. Rahman, and J. Noor, "Low-cost, low-power, and low-compute based ecg monitoring systems: Comparative analysis and beyond," in *2022 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*, pp. 401–407, IEEE, 2022.

[256] M. M. T. Nur, S. S. Dola, A. K. Banik, T. Akhter, N. Hossain, A. A. Al Islam, and J. Noor, "Speaker identification through gender detection," in *2022 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*, pp. 181–188, IEEE, 2022.

[257] M. M. Ahasan, M. Fahim, H. Mazumder, N. E. Fatema, S. M. Rahman, A. A. Al Islam, and J. Noor, "Classification of respiratory diseases and covid-19 from respiratory and cough sounds," in *2022 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*, pp. 707–714, IEEE, 2022.

[258] S. H. Nova, S. M. Quader, S. D. Talukdar, M. R. Sadab, M. S. Sayeed, A. A. Al Islam, and J. Noor, "Iot based parking system: Prospects, challenges, and beyond," in *2022 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*, pp. 393–400, IEEE, 2022.

[259] M. S. Mustafa, J. F. K. Lisa, and J. N. Mukta, "Design and implementation of wireless iot device for women's safety," in *Proceedings of the 9th International Conference on Networking, Systems and Security*, pp. 41–52, 2022.

[260] F. F. Khan, N. M. Hossain, M. N. H. Shanto, S. B. Anwar, and J. Noor, "Mitigating ddos attacks using a resource sharing network," in *Proceedings of the 9th International Conference on Networking, Systems and Security*, pp. 1–11, 2022.

[261] R. F. Aunindita, M. S. Misbah, S. B. Joy, M. A. Rahman, S. I. Mahabub, and J. N. Mukta, “Use of machine learning and iot for monitoring and tracking of livestock,” in *2022 25th International Conference on Computer and Information Technology (ICCIT)*, pp. 815–820, IEEE, 2022.

[262] M. I. Miah, J. C. Gope, A. D. Nath, A. J. Nain, F. N. Mitu, and J. Noor, “Advanced waterway transport system based on internet of things (iot): A novel approach,” in *2022 25th International Conference on Computer and Information Technology (ICCIT)*, pp. 1–6, IEEE, 2022.

[263] T. M. Monsaif, O. F. Alif, S. D. Amarth, T. A. Sadman, and J. Noor, “A novel approach to reduce air pollution through machine learning based pm2. 5 prediction,” in *2022 4th International Conference on Sustainable Technologies for Industry 4.0 (STI)*, pp. 1–8, IEEE, 2022.

[264] G. Aarts, P. Giudice, and E. Seiler, “Localised distributions and criteria for correctness in complex langevin dynamics,” *Annals of Physics*, vol. 337, pp. 238–260, Oct 2013.