Cloud based Secure Data Transfer using Exchange of Cryptographic Keys

A major project report submitted in partial fulfillment of the requirement for the award of degree of

Bachelor of Technology
in
Computer Science & Engineering / Information Technology

Submitted by
Utsav 201292
Devansh Chaudhary 201461

Under the guidance & supervision of
Dr. Hari Singh

Department of Computer Science & Engineering and Information Technology
Jaypee University of Information Technology, Waknaghat, Solan - 173234 (India)

CERTIFICATE

This is to certify that the work presented in this report entitled **'Cloud based Secure Data Transfer using Exchange of Cryptographic Keys'** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science & Engineering / Information Technology** submitted in the Department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology, Waknaghat is an authentic record of my own work carried out over a period from August 2023 to May 2024 under the supervision of **Dr. Hari Singh** (Assistant Professor (SG), Department of Computer Science & Engineering and Information Technology).

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Devansh Chaudhary	Utsav
201461	201292

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

Dr. Hari Singh
Assistant Professor (SG)
Department of Computer Science & Engineering and Information Technology
Dated:

Candidate's Declaration

We hereby declare that the work presented in this report entitled **'Cloud based Secure Data Transfer using Exchange of Cryptographic Keys'** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science & Engineering / Information Technology** submitted in the Department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology, Waknaghat is an authentic record of my own work carried out over a period from August 2023 to May 2024 under the supervision of **Dr. Hari Singh** (Assistant Professor (SG), Department of Computer Science & Engineering and Information Technology).

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Devansh Chaudhary	Utsav
201461	201292

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

Dr. Hari Singh
Assistant Professor (SG)
Department of Computer Science & Engineering and Information Technology
Dated:

ACKNOWLEDGEMENT

Firstly, I express my heartiest thanks and gratefulness to almighty God for His divine blessing makes it possible to complete the project work successfully. I am really grateful and wish my profound indebtedness to Supervisor **Dr. Hari Singh**, Assistant Professor (SG), Department of CSE Jaypee University of Information Technology, Wakhnaghat. Deep Knowledge & keen interest of my supervisor in the field of "Cloud Computing" and "Information Security" to carry out this project. His endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism, valuable advice, reading many inferior drafts and correcting them at all stages have made it possible to complete this project. I would like to express my heartiest gratitude to Dr. Hari Singh, Department of CSE, for his kind help to finish my project. I would also generously welcome each one of those individuals who have helped me straightforwardly or in a roundabout way in making this project a win. In this unique situation, I might want to thank the various staff individuals, both educating and non-instructing, which have developed their convenient help and facilitated my undertaking.

Finally, I must acknowledge with due respect the constant support of my parents.

Devansh Chaudhary

201461

Computer Science & Engineering and Information Technology

Jaypee University of Information Technology, Waknaghat

Utsav

201292

Computer Science & Engineering and Information Technology

Jaypee University of Information Technology, Waknaghat

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

S. No.	Title
1	AMQP: Advanced Messaging Queue Protocol
2	RESTful API: Representational State Transfer Application Programming Interface
3	S3: Simple Storage Service
4	FFmpeg: Fast Forward Moving Picture Experts Group
5	AWS: Amazon Web Services

ABSTRACT

The secure and effective movement of data to and from cloud environments is a top priority for organisations across sectors in the present era of digital transformation. This large project presents a comprehensive solution that combines rigorous security measures, data compression techniques, Advanced Message Queuing Protocol, and Kubernetes cluster scalability.

The project utilises encryption techniques and authentication procedures to provide a safe basis for data transit. These safeguards are intended to protect sensitive information while it travels across the cloud infrastructure, maintaining its confidentiality and integrity. This security architecture is vital for safeguarding critical data from potential attacks and unauthorised access.

Aside from security concerns, the initiative prioritises bandwidth utilisation and overall transmission efficiency. Data compression technologies are combined to lower video file data capacity. This method reduces transfer times and costs, resulting in a more resource-efficient cloud architecture.

To improve scalability and manageability, the project also makes use of the capabilities of Kubernetes clusters. Kubernetes orchestrates containerized applications, allowing for smooth resource scaling based on demand, ensuring that data transport stays efficient in dynamically changing contexts. Integration with Kubernetes clusters enables fault tolerance and high availability, both of which are crucial for mission-critical data transfer activities.

This combination of secure data transmission, compression, and Kubernetes cluster integration provides a comprehensive solution to core difficulties connected with cloud-based data sharing. This project intends to contribute to the development of cloud infrastructures that are durable, efficient, and secure. It promotes a technical environment in which organisations may confidently exploit the benefits of cloud computing while maintaining data integrity and security.

CHAPTER 1: INTRODUCTION

1.1 INTRODUCTION

The ever-increasing reliance on cloud computing services has necessitated the development of efficient and reliable data transfer methodologies. Traditional approaches to cloud data sharing often struggle with handling large file sizes, optimizing compression techniques, and scaling systems to meet growing demands. This research proposes a novel solution that leverages state-of-the-art technologies, including Kubernetes and RabbitMQ Advanced Message Queuing Protocol (AMQP), to address these challenges and provide a scalable, secure, and high-performance cloud data sharing framework.

Cloud computing is the on-demand delivery of computing services such as servers, storage, databases, networking, software, and analytics [1]. Rather than keeping files on a proprietary hard drive or local storage device, cloud-based storage makes it possible to save remotely. Cloud computing is a popular option for people and businesses, allowing for cost savings, increased productivity, speed and efficiency, performance, and security [2].

There are numerous methods for deploying Kubernetes clusters that differ in customization options, supported clouds, and costs. Some methods have existed since Kubernetes' inception, while others like AWS EKS were introduced later. Existing comparisons between different Kubernetes deployment methods either rely on theoretical information from technical documentation, are presented informally as blog posts through non-academic processes, or consider only a basic vanilla cluster configuration [3].

Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications. It allows developers to focus on building and deploying their applications without worrying about the underlying infrastructure[4]. Kubernetes uses a declarative approach to managing applications, where users specify desired application states, and the system maintains them. It also provides robust tools for monitoring and managing applications, including self-healing mechanisms for automatic failure detection and recovery. Overall, Kubernetes offers a powerful and flexible solution for managing containerized applications in production environments [5].

The advanced message queuing protocol (AMQP) working group's goal is to create an open standard for an interoperable enterprise-scale asynchronous messaging protocol. AMQP is finally addressing the lack of enterprise messaging interoperability standards. This relatively simple yet compellingly powerful enterprise messaging protocol is thus poised to open up a bright new era for enterprise messaging [6].

The Advanced Message Queuing Protocol (AMQP) aims to create an open standard for interoperable, enterprise-scale asynchronous messaging. This study implemented AMQP 1.0 over QUIC and conducted extensive testing using the NS3 network simulator. The results demonstrated significant improvements in communication time and start-up latency, enabling faster and more reliable connections [7].

RabbitMQ is an open-source messaging system that allows users to integrate applications together by using messages and queues. RabbitMQ implements the AMQP. The underlying RabbitMQ server is written in the Erlang programming language which was originally designed for the telecoms industry by Ericsson. Erlang supports distributed, fault-tolerant applications and is, therefore, an ideal language to use to build a message queuing system [8].

Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. Customers of all sizes and industries can use Amazon S3 to store and protect any amount of data for a range of use cases, such as data lakes, websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics. Amazon S3 provides management features so that you can optimize, organize, and configure access to your data to meet your specific business, organizational, and compliance requirements [9].

Streaming is also more efficient in terms of bandwidth and storage requirements, as only the portions of the video being processed need to be transmitted and temporarily buffered, instead of storing the complete file locally. This streaming approach is particularly beneficial for time-sensitive applications like live video analytics, where insights need to be derived from the video data with minimal latency. Additionally, streaming seamlessly supports processing of extremely long videos or even infinite streams of data, which would be impractical to fully download and store. By leveraging video streaming, compute resources can be optimized to extract value from video content as it is captured and transmitted, enabling a more agile and responsive video processing pipeline [10].

The proposed approach employs RabbitMQ for effective management of compression requests, mitigating the risk of server overload and ensuring reliable data processing. By leveraging Kubernetes, compression worker nodes can be dynamically scaled based on real-time demand, optimizing resource utilization and ensuring cost-effective operation. Furthermore, the solution introduces a streaming mechanism for retrieving file byte data from S3, eliminating the need for local storage and significantly reducing processing time, especially for large files. This innovative approach not only addresses existing limitations but also paves the way for a paradigm shift in cloud data transfer methodologies.

1.2 PROBLEM STATEMENT

The cloud data transfer market is characterised by ongoing problems that need a paradigm shift in existing approaches. Traditional approaches frequently fail to leverage compression techniques effectively, resulting in major challenges with scalability, reliability, and resource management.

1.2.1 INEFFECTIVE COMPRESSION UTILIZATION

In the landscape of contemporary cloud data transfer methodologies, a pervasive issue emerges in the suboptimal utilization of compression techniques. This inefficiency results in a less-than-ideal reduction of large file volumes, manifesting in a two-fold consequence. Firstly, the transfer times for these voluminous files are needlessly prolonged, impacting the efficiency of data transfer processes. Secondly, and perhaps more critically, there is an increased demand for storage resources as the inadequately compressed files occupy more space than necessary.

The ramifications of this ineffective compression utilization extend beyond mere inconveniences. They pose a significant bottleneck, hindering the seamless flow of information across cloud platforms. As organizations grapple with the exponential growth of data volumes, this inefficiency becomes a pronounced challenge, impeding the swift and resource-efficient transfer of crucial information in cloud-based environments. Addressing this inefficiency becomes paramount for optimizing data transfer processes and ensuring that the infrastructure can handle the escalating demands of modern data management.

1.2.2 SCALABILITY AND RELIABILITY CONCERNS

An ongoing concern in the landscape of modern cloud data transport procedures is the underutilization of compression algorithms. This inefficiency leads to a less-than-ideal decrease of huge file sizes, which has two consequences. For starters, the transfer durations for these large files are unnecessarily long, reducing the efficiency of data transmission procedures. Second, and possibly more importantly, there is a greater demand on storage resources since improperly compressed files use up more space than necessary.

The consequences of inadequate compression usage go beyond minor annoyance. They are a major impediment in the smooth flow of information among cloud platforms. As organisations battle with the exponential expansion of data quantities, this inefficiency becomes a significant barrier, preventing the timely and cost-effective transmission of critical information in cloud-based settings. Addressing this inefficiency is critical for optimising data transfer operations and ensuring that the infrastructure can support the ever-increasing needs of contemporary data management.

1.2.3 RESOURCE CONSTRAINTS AND SERVER CRASHES

In the context of cloud data transport, the complicated interplay between the resource-intensive nature of compression activities and the general stability of servers emerges as a crucial problem. The severe computing needs inherent in compression methods generate significant concerns regarding server stability, especially when resource limits are present. When server resources are insufficient to manage the computational demand, the threat of crashes during compression procedures becomes a critical point of failure.

This important moment not only jeopardises continuous data transfers, but also raises the spectre of data loss and system outage. The insecurity of present approaches is exacerbated by the instability of servers under the burden of compression operations. Addressing these resource limits becomes critical for assuring continuous data flow, protecting against data loss, and preserving the integrity of the cloud data transfer system. In order to mitigate these risks and reinforce the system against unexpected failures, strategies for efficient resource allocation, dynamic scaling, and strong error recovery methods are critical.

1.2.4 LATENCY IN COMPRESSION PROCESSING

While RESTful APIs for compression requests provide a handy interface for user interactions, they come with a trade-off in the shape of data transmission delay. Waiting delays are introduced by the sequential nature of RESTful API requests, notably during the completion of compression activities, which greatly influences the entire transfer time. This delay undermines the projected efficiency improvements from using compression methods.

The system encounters delays in responding to subsequent requests because users are forced to wait until the compression process is complete before proceeding with additional activities. This delay degrades not just the user experience but also the overall efficiency of the data transfer process. In an age when speed and responsiveness are critical, the delay induced by RESTful API-based compression requests becomes a significant disadvantage.

To solve this, alternate techniques that allow for asynchronous compression task processing or the use of more responsive protocols might be beneficial. By reducing latency in compression processing, the system may not only improve overall data transfer efficiency, but also provide a more fluid and user-friendly experience that meets the demands of current cloud data transfer settings.

1.2.5 LACK OF FAULT TOLERANCE

One obvious flaw in many existing approaches to cloud data transport is the lack of effective fault tolerance measures, particularly during the complex compression process. The compression process, as a critical phase in the data transmission workflow, necessitates robustness in the face of unanticipated problems. Unfortunately, the current absence of adequate fault tolerance methods reveals a vulnerability with far-reaching implications.

When a server crashes or a node fails during the compression process, the lack of adequate fault tolerance methods becomes a major source of worry. Without proper procedures in place to gracefully handle failures, the system is vulnerable to data loss, jeopardising the whole data transmission process's dependability. This flaw not only jeopardises the integrity of the sent data but also increases the possibility of interruptions to current data transfers.
Addressing the system's lack of fault tolerance becomes critical for assuring its resilience in the face of unforeseen events. Data replication, automated failover systems, and robust error

recovery procedures, for example, can considerably improve the system's capacity to tolerate disturbances. As a result, the system obtains a degree of robustness that not only protects against data loss in the case of server breakdowns, but also strengthens the general dependability of the cloud data transfer process. As organisations rely more on frictionless and trustworthy data transmission technologies, incorporating fault tolerance techniques becomes a strategic need for maintaining data integrity in transit.

In conclusion, the current issues with cloud data transmission protocols underscore the crucial need for a comprehensive and novel solution. This project seeks to address these issues front on by presenting a methodology that not only optimises compression techniques but also addresses scalability, dependability, and resource management issues, ushering in a new era of safe and efficient cloud data transport.

1.3 OBJECTIVES

This unique project's principal goals are multifarious, with the goal of overcoming the problems inherent in present cloud data transmission technologies. The complete set of goals is intended to improve the data transfer process's efficiency, scalability, dependability, and resource management, resulting in a secure and optimised solution.

1.3.1 OPTIMIZED COMPRESSION TECHNIQUES

The primary goal is to create a robust system that uses Compression Techniques to successfully decrease huge file sizes during cloud data transport. The project aims to dramatically improve the compression process by using new algorithms and techniques, lowering transfer times and lessening the need for storage resources.

1.3.2 SCALABILITY AND RELIABILITY ENHANCEMENT

A critical goal is to address the scalability and reliability difficulties associated with compression tasks. The project intends to distribute compression requests to dedicated queues on different servers using RabbitMQ AMQP. This not only reduces the chance of server breakdowns, but also assures a dependable and fluid data transmission experience, even when several users make concurrent requests.

1.3.3 RESOURCE MANAGEMENT WITH KUBERNETES

The project aims to dynamically distribute and grow compression worker nodes based on the real-time length of the compression request queue, with Kubernetes serving as a major enabler. This dynamic resource management guarantees that server resources are used optimally, reducing crashes due to computational intensity, and optimising overall system performance.

1.3.4 STREAMLINED DATA PROCESSING

An important goal is to use byte streaming to receive file byte data from S3, as opposed to typical approaches that include waiting for file downloads, local storage, and future uploads. This innovation not only saves processing time but also the requirement for substantial server storage space, which is especially useful when dealing with files with greater volumes than accessible local storage.

1.3.5 ROBUST ERROR HANDLING MECHANISMS

It is critical to design a system with strong error handling methods. The project intends to develop a requeuing mechanism in the case of a node failure during request processing, guaranteeing that the request is eventually executed. This improves the system's overall fault tolerance, which contributes to the dependability of the cloud data transmission process.

Finally, the project's objectives comprise a complete framework targeted at changing the landscape of cloud data transport. The project aims to provide a secure, efficient, and reliable solution to the challenges posed by modern cloud data transfer methodologies by optimising compression techniques, improving scalability and reliability, dynamically managing resources, streamlining data processing, and implementing robust error handling mechanisms.

1.4 SIGNIFICANCE AND MOTIVATION OF THE PROJECT WORK

This research is significant because it has the potential to revolutionise the efficiency, reliability, and security of cloud data transport. The motive for embarking on this novel endeavour is to solve the critical difficulties confronting present techniques while also unleashing new possibilities in the field of data management.

1.4.1 ENHANCING DATA TRANSFER EFFICIENCY

One of the key motivators for this research is the critical need to improve the efficiency of cloud data transport. Traditional techniques, which are hampered by inefficiencies in compression utilisation and processing delay, frequently result in lengthy transfer times. The project seeks to drastically reduce transfer times by optimising compression techniques and using a streaming approach for data processing, resulting in a more streamlined and responsive data transfer experience.

1.4.2 ENSURING SCALABILITY AND RELIABILITY

In the context of current data transmission demands, the project's emphasis on scalability and dependability is critical. The usual issues of server failures, resource restrictions, and the possibility of data loss during concurrent compression requests need a strong solution. The project secures not only the system's scalability but also its resilience in the face of unforeseen interruptions by integrating RabbitMQ AMQP and Kubernetes, giving users with a dependable and uninterrupted cloud data transfer service.

1.4.3 RESOURCE OPTIMIZATION AND COST EFFICIENCY

Efficient resource management is a key component of this project, which is driven by the need to optimise server resources and decrease operational costs. The system guarantees that resources are used efficiently by dynamically assigning and scaling compression worker nodes depending on real-time demand, reducing wasteful server failures and lowering infrastructure costs. This resource optimisation is critical in today's world, when cost efficiency and sustainability are top priorities for businesses.

1.4.4 STREAMLINING DATA PROCESSING

The incentive for using a streaming technique for data processing is the potential for it to revolutionise how files are handled during transit. The streaming strategy, as opposed to traditional approaches that entail many stages such as downloading, local storage, and later uploading, considerably decreases processing time. This is especially useful for organisations dealing with massive file volumes that exceed accessible local storage, making the project's technique not only efficient but also flexible to a variety of storage circumstances.

1.4.5 ROBUSTNESS AND RELIABILITY ASSURANCE

The project's emphasis on strong error handling techniques is motivated by the significance of assuring the integrity and dependability of the data transmission process. The system improves its fault tolerance by incorporating requeuing techniques in the event of node failures. This resilience means that, even in difficult conditions such as server restarts or outages, every compression request is finally executed, adding to the overall stability of the cloud data transmission system.

In short, the relevance and motivation for this research stem from its potential to revolutionise cloud data transmission procedures. By tackling the inefficiencies, problems, and constraints of existing methodologies, the project aims to produce a solution that not only meets but surpasses the expectations of efficiency, scalability, dependability, and resource optimisation in the changing environment of cloud computing.

1.5 ORGANIZATION OF PROJECT REPORT

The project report is organised in a methodical manner, encompassing various stages of research and development. Each chapter contributes in its own way to offering a comprehensive knowledge of the project, including its context, implementation, and consequences.

Chapter 1: Introduction

The opening chapter provides a comprehensive summary of the project, covering the history, motivation, aims, and technique used. It highlights the limitations of modern cloud data transport and the novel technique provided to overcome these concerns.

Chapter 2: Literature Review

The second chapter undertakes a thorough literature review to investigate and analyse existing methods to cloud data transport and compression techniques. This evaluation outlines the project's background, identifies shortcomings in conventional approaches, and emphasises the necessity for the suggested creative solution.

Chapter 3: System Development

Chapter 3 delves into the project's heart by revealing the complexities of system development. It delves into the methodology's implementation, including the integration of RabbitMQ AMQP, the use of Kubernetes for resource management, and the use of a streaming approach for efficient data processing. This chapter also provides a detailed technical overview of how the proposed system works and the reasoning behind design decisions.

Chapter 4: Testing

The fourth chapter focuses on the thorough testing that was performed to evaluate the functionality, reliability, and efficiency of the implemented system. It describes the testing approaches used, such as unit testing, integration testing, and system testing. The results of these tests are provided and analysed to assure the resilience and efficacy of the established system.

Chapter 5: Results and Evaluation

Chapter 5 is devoted to presenting the testing findings and providing a thorough review of the system's performance. This section offers quantitative and qualitative assessments that demonstrate the gains made in data transmission efficiency, scalability, and dependability. The outcomes are evaluated in light of the project's objectives and expectations.

Chapter 6: Conclusions and Future Prospects

The last chapter summarises the important findings, draws conclusions based on the data and insights gathered, and examines the project's larger ramifications. It also highlights the future scope of the study, identifying prospective opportunities for additional research and development in the realm of cloud data transfer.

In accordance with this organised framework, the project report seeks to give a clear and complete narrative, taking the reader through the path of ideation, development, testing, and assessment of the revolutionary cloud data transfer approach.

CHAPTER 2: LITERATURE SURVEY

2.1 OVERVIEW OF RELEVANT LITERATURE

When it comes to moving data in the cloud, many current methods don't use a smart way to shrink the size of big files. Even those that try often run into problems with how well they can handle lots of data and keep everything working smoothly.

The way people usually compress data can be really hard on the computer, and it might even make the server crash if it doesn't have enough power. Also, using a particular kind of tool called RESTful API to ask the computer to compress things means we have to wait until it's all done. If lots of people ask the computer to compress things at the same time, it could crash, and we might lose some data.

Our solution to these issues is pretty cool. Instead of the usual way, we use a system called RabbitMQ to ask another computer to compress things for us. We've set up special worker computers using Kubernetes that are always ready to compress files. The number of these worker computers changes based on how many compression requests there are: more requests mean more workers, and when things calm down, we use fewer workers. If a worker computer has trouble, we try the request again, and the worker computer gets a fresh start.

Inside the worker computer, we use a clever method to get pieces of data from the file we want to compress. This helps us speed up the whole process and saves us from needing a lot of space on the computer. This is especially helpful when dealing with really big files that won't fit in our computer's storage.

Once the compression is done, we organize the results and update a list that keeps track of where the compressed file is. We also send a message to the person who asked for the compression, letting them know it's all done.

By using RabbitMQ and Kubernetes, we make sure that even if the computer crashes and restarts, it will still finish compressing everything. Kubernetes also helps us manage the worker computers efficiently, adding more when needed and reducing them when things slow down. This way, we use our resources wisely and save money on running the computers. Our approach is like having a smart team that knows how to handle lots of data without causing problems.

The authors compared two modes of communication between microservices, RabbitMQ that is characterized by asynchronous messaging, message queuing, publish-subscribe, scalability and RESTful APIs characterized by simplicity, statelessness, widespread adoption, platform-independent. They found that the RESTful APIs suffer from limited asynchronous support, potential message loss, network overhead in certain cases, tight coupling in synchronous communication while RabbitMQ requires infrastructure setup, learning curve, complexity in synchronous communication but has the benefit of scalability, reliability and asynchronous communication [11].

The study focuses on a Kubernetes-based monitoring platform for dynamic cloud resource provisioning, highlighting its methodology centered on real-time monitoring within Kubernetes. Notable advantages include efficiency and scalability, but challenges such as complexity and a steep learning curve are evident. Recommendations for improvement include enhancing automation, integrating machine learning, improving usability, and bolstering security. These enhancements promise to optimise resource management and performance in dynamic cloud environments, aligning with evolving technological demands [12].

The research introduces an estimation-based dynamic load-balancing algorithm tailored for heterogeneous grid computing environments. Methodologically, it focuses on crafting this algorithm to optimize load distribution. Notable benefits include efficient resource utilization, dynamic adaptation, and heightened performance and reliability. However, challenges like the need for precise estimation and implementation complexity exist. To advance, refining estimation accuracy, simplifying implementation, and integrating machine learning for resource allocation are suggested. These improvements promise to bolster the algorithm's efficacy, paving the way for more efficient load balancing in diverse computing environments [13].

The study presents an innovative data-sharing scheme for mobile devices in cloud computing, where encryption and decryption tasks are executed on the cloud server instead of local devices. This methodology offers speed advantages for mobile devices. However, concerns arise regarding reduced security and increased server-side computational load. To progress, enhancing scalability, fortifying security measures, and optimizing server-side computation are imperative. Additionally, leveraging heavy compression computation on the server side enables lightweight functionality for mobile applications, further enhancing efficiency [14].

The paper examines Docker Cluster Management for the Cloud through a multi-step methodology involving use case analysis, requirement derivation, tool selection based on documentation, and rigorous testing. Docker's lightweight and efficient containerization technology is a notable advantage. However, existing Docker cluster management tools lack full integration and fail to meet all container management requirements in cloud environments. Improvement opportunities lie in developing a more integrated and comprehensive solution, specifically addressing the limitations and challenges of container management. This research emphasizes Docker's significance in facilitating Kubernetes deployment [15].

Another study proposes a methodology for cloud application deployment that emphasizes transient failure recovery. It deploys redundant application instances across multiple cloud servers, dynamically adjusts resources through auto-scaling, continuously monitors health with alerts, ensures data integrity, and employs circuit breakers for fault tolerance. Notable advantages include high availability, improved performance, reduced downtime, data integrity assurance, and cost-efficiency through optimal resource utilization. However, the approach introduces increased complexity, potential resource overhead, and additional development effort. Areas for further improvement include predictive failure analysis, streamlined implementation, enhanced automation, and integration of advanced recovery techniques like checkpointing or replication. Addressing these aspects could enhance the resilience, reliability, and efficiency of cloud application deployments, especially in failure-prone environments [16].

A bibliometric study analyzed research trends and patterns related to Kubernetes as a standard container orchestrator. The methodology involved using the Bibliometrix R package to extract and analyze bibliometric data from the Web of Science database. This approach provides a comprehensive quantitative overview of the research landscape, identifying current trends, influential works, and potential future directions.

A notable advantage is the ability to systematically map and understand the evolution of this field. However, limitations include being confined to the Web of Science database, potentially missing relevant research from other sources, and focusing primarily on quantitative bibliometric indicators. Areas for improvement could involve combining bibliometric analysis with qualitative methods, expanding the data sources to include other databases, preprints, and conference proceedings. This would yield a more holistic understanding of Kubernetes research while compensating for the limitations of solely relying on bibliometric data from a single database [17].

This proposed Kubernetes system architecture by the author with a proactive custom autoscaler using a Bi-LSTM deep learning model is proposed. The Bi-LSTM accurately predicts future workloads from historical data, outperforming LSTM and ARIMA models. The planning phase mitigates oscillations and efficiently removes surplus pods during decreasing workloads for burst handling. Evaluations demonstrate the Bi-LSTM's superior accuracy, prediction speed, resource provisioning accuracy, and elastic scaling capabilities compared to existing approaches. The architecture outperforms Kubernetes' default autoscaler in dynamic resource scaling [18].

S. No.	Paper Title	Journal/ Conference (Year)	Tools/ Techniques /Dataset	Pros	Cons
1	An Estimation-Based Dynamic Load Balancing Algorithm. [13]	Journal of Grid Computing / 2023	Simulation experiments based on GridSim toolkit	Achieved better productivity, high average resource utilization.	Evaluated using simulation experiments; did not consider the impact of network latency
2	File Transfer on Cloud using Diffie-Hellman Key Exchange in Conjunction with AES Encryption [19]	National College of Ireland (2023)	Diffie-Hellman Key Exchange, AES Encryption	Enhanced data security during cloud-based file transfers.	Limited scalability for large files
3	An efficient and secure compression technique for data protection using burrows-wheeler transform algorithm [20]	Heliyon / 2023	Burrows-Wheeler Transform Algorithm	Efficient data compression and protection.	Limited application for specific data types; might have varying effectiveness for different file formats.
4	An Efficient Algorithm for Secure Key Management in Cloud Environment [21]	IEEE ICONSTEM (2023)	Secure Key Management Algorithm	Efficient algorithm for secure key exchange in cloud environments, addressing key generation, distribution, and storage	Further validation in diverse cloud environments

S. No.	Paper Title	Journal/ Conference (Year)	Tools/ Techniques /Dataset	Pros	Cons
5	Kubernetes as a Standard Container Orchestrator - A Bibliometric Analysis [17]	Journal of Cloud Computing / 2022	Bibliometrix	The analysis identified the main research hotspots like Auto Scalling in Kubernetes	Kubernetes not suitable for low users applications
6	Secure Data Transfer Based on Cloud Computing [22]	IRJET (Mar 2022)	DPaaS, Diffie Hellman key exchange	Improved Security, reduced risk and efficacy performance	Resource Constraints; Security Trade-offs
7	Password-Authenticated Key Exchange from Group Actions [23]	Annual International Cryptology Conference / 2022	Group Actions in Cryptography	Secure password-authenticated key exchange protocols.	Requires additional steps in the authentication process; may have higher computational overhead.
8	An efficient and secure data sharing scheme for mobile devices in cloud computing [14]	Journal of Cloud Computing / 2020	Elliptic Curve Cryptography - key base encryption algorithm	Achieves lightweight computation on mobile devices.	High computation task make the device run out of memory and led to app crash.
9	A Comparison between RabbitMQ and RESTful API for Communication between	IRJET / 2020	RabbitMQ, RESTful API, Database operations,	AMQP is a better option as it improves the speed and	Performance of RabbitMQ and RESTful API may vary depending on

S. No.	Paper Title	Journal/ Conference (Year)	Tools/ Techniques /Dataset	Pros	Cons
	Microservices Web Application [11]		AMQP protocol	efficiency of micro-service	the specific requirements
10	Cloud application deployment with transient failure recovery [16]	Journal of Grid Computing / 2018	AURA, a cloud deployment system	AURA helps to minimise failures and does not increase time to deploy	Slow down deployment
11	A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning [12]	IEEE GLOBECOM / 2017	Kubernetes CLI (kubectl), Apache JMeter, Docker	Dynamically adjust resource provisioning based on CPU and memory utilization.	Tested on a single application; Increased complexity
12	Docker Cluster Management for the Cloud: A Comprehensive Survey and Solution [15]	Journal of Grid Computing / 2016	FluentD (data collection), Graylog 2	Docker led to easy deployment and management	Not suitable for all use cases since it is time-consuming to set up.

Table 1: Literature Survey

2.2 KEY GAPS IN THE LITERATURE

The presented text reveals several gaps in the existing literature on cloud data transfer methodologies. Many current approaches struggle with the efficient reduction of large file sizes, leading to increased transfer times and a greater demand for storage resources. Additionally, scalability and reliability concerns arise due to the computational intensity of compression tasks, risking server crashes, especially in resource-constrained environments. The reliance on RESTful API for compression exacerbates these challenges, introducing waiting times and escalating the likelihood of crashes, particularly under concurrent user requests.

The proposed solution introduces a novel approach using RabbitMQ and Kubernetes, offloading compression tasks to dedicated worker computers and dynamically adjusting their numbers based on request volumes, addressing scalability and reliability issues. This innovative method also minimizes the risk of server crashes, efficiently manages computational demands, and enhances fault tolerance. Furthermore, the approach introduces a space-efficient data processing method within worker computers, optimizing storage space utilization.

Real-time updates and notifications, facilitated by RabbitMQ, ensure users are promptly informed upon completion of compression tasks, enhancing the overall user experience and system transparency. In essence, the proposed methodology not only addresses these identified gaps but sets a forward-looking standard for efficient and reliable cloud data transfer.

CHAPTER 3: SYSTEM DEVELOPMENT

3.1 REQUIREMENTS AND ANALYSIS

A detailed examination of current ways to secure cloud data transfer finds a common shortcoming in successfully using compression techniques to meet the constraints posed by huge file sizes. Despite embracing compression, many previous systems struggle with scalability and reliability difficulties. This research emphasises the vital importance of a paradigm shift in tackling these difficulties in order to assure a more efficient and trustworthy cloud data transport infrastructure.

The computational intensity associated with compression job execution appears as a major challenge. The possibility of server failures looms large, especially when the server lacks the requisite capabilities to meet the processing demands. This vulnerability not only jeopardises the system's stability, but it also raises worries about potential data loss, which is crucial in any data-centric application.

The use of RESTful API for compression requests adds another degree of complication. Because of the sequential nature of RESTful API, it is necessary to wait for the compression process to be completed before replying to user queries. This not only affects system responsiveness but also increases the chance of server failures, especially in cases with numerous users submitting compression requests at the same time. The combination of these difficulties creates a significant risk to the integrity and stability of the data transmission process.

The combination of Kubernetes and RabbitMQ emerges as a strategic requirement to handle these complexities and improve the overall system design. Kubernetes' dynamic scaling features protect against server problems caused by computational intensity. Kubernetes guarantees optimal resource utilisation by automatically distributing compression worker nodes based on the length of the compression request queue. This dynamic scaling approach not only reduces the chance of server breakdowns, but it also promotes scalability, allowing the system to handle variable workloads with ease.

Simultaneously, RabbitMQ delivers a queuing system that decouples compression requests from direct RESTful API calls via its Advanced Message Queuing Protocol (AMQP). This deliberate decoupling not only avoids the need to wait for the compression process to complete before replying to users, but it also guarantees that requests are processed in an ordered manner. The queuing mechanism speeds up the processing of requests even if the server restarts, contributing to the system's stability and fault tolerance.

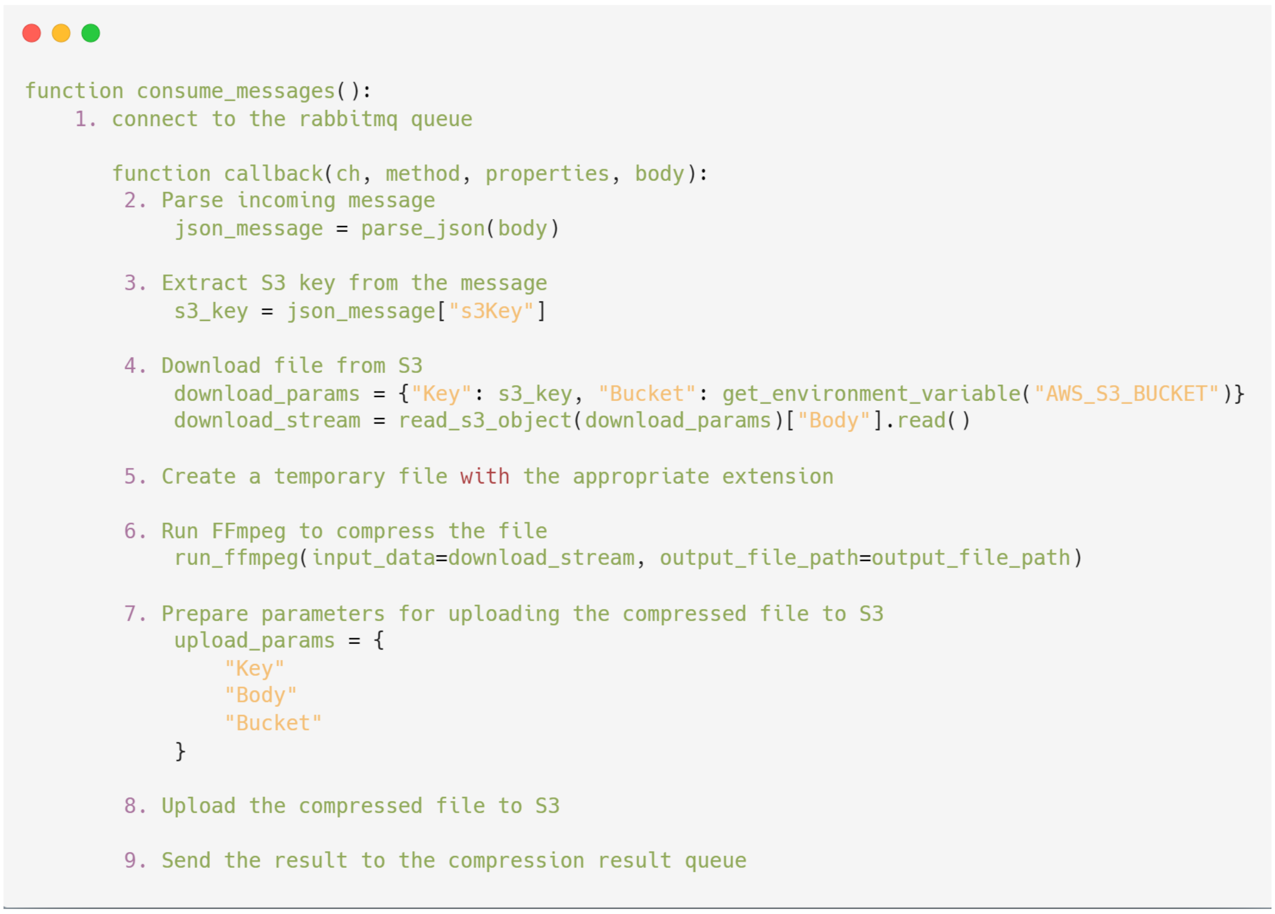
In summary, the requirements analysis emphasises the need for a sophisticated solution that handles the issues given by enormous file quantities, processing intensity, and probable server breakdowns. The combination of Kubernetes with RabbitMQ is a critical step towards developing a strong and scalable cloud data transmission system that not only overcomes existing shortcomings but also establishes a precedent for more effective and dependable data processing in cloud settings.

3.2 PROJECT DESIGN AND ARCHITECTURE

Many contemporary approaches to cloud data transfer lack effective utilization of compression techniques for the reduction of large file volumes. Furthermore, among those that do employ compression, a deficiency in scalability and reliability is frequently observed.

The implementation of compression tasks is computationally intensive, posing the risk of server crashes in instances where the server lacks adequate resources. The use of RESTful API for compression requests exacerbates this issue, as it necessitates waiting until the compression process is complete. Concurrent submission of compression requests by multiple users further amplifies the risk of server crashes, potentially resulting in data loss.
Our approach involves using RabbitMQ Advanced Message Queuing Protocol (AMQP) to route compression requests to a different server's dedicated queue [14]. We used Kubernetes to construct compression worker nodes that continually check the queue for incoming requests. The length of the compression request queue determines the dynamic allocation of nodes: as the queue length rises, new nodes are deployed to handle requests, and when the queue length reaches zero, excess nodes are scaled down. If a node fails while processing a request, the request is requeued and the node is restarted.

We use a streaming technique within the compression node to obtain file byte data from Amazon S3. As depicted in Figure 1, the compression algorithm analyzes the streaming byte data, and the resultant compressed file byte stream is transmitted to the S3 uploader.



```
function consume_messages():
    1. connect to the rabbitmq queue

       function callback(ch, method, properties, body):
        2. Parse incoming message
            json_message = parse_json(body)

        3. Extract S3 key from the message
            s3_key = json_message["s3Key"]

        4. Download file from S3
            download_params = {"Key": s3_key, "Bucket": get_environment_variable("AWS_S3_BUCKET")}
            download_stream = read_s3_object(download_params)["Body"].read()

        5. Create a temporary file with the appropriate extension

        6. Run FFmpeg to compress the file
            run_ffmpeg(input_data=download_stream, output_file_path=output_file_path)

        7. Prepare parameters for uploading the compressed file to S3
            upload_params = {
                "Key"
                "Body"
                "Bucket"
            }

        8. Upload the compressed file to S3

        9. Send the result to the compression result queue
```

Figure 1: Compression Node Pseudo Code

Unlike the conventional strategy, which entails waiting for a file download, storing locally, compressing, and then uploading to S3, as seen in Figure 2, our streaming byte data method greatly decreases processing time and the requirement for considerable server storage space. This is very useful when compressing files with bigger sizes than the available local storage.

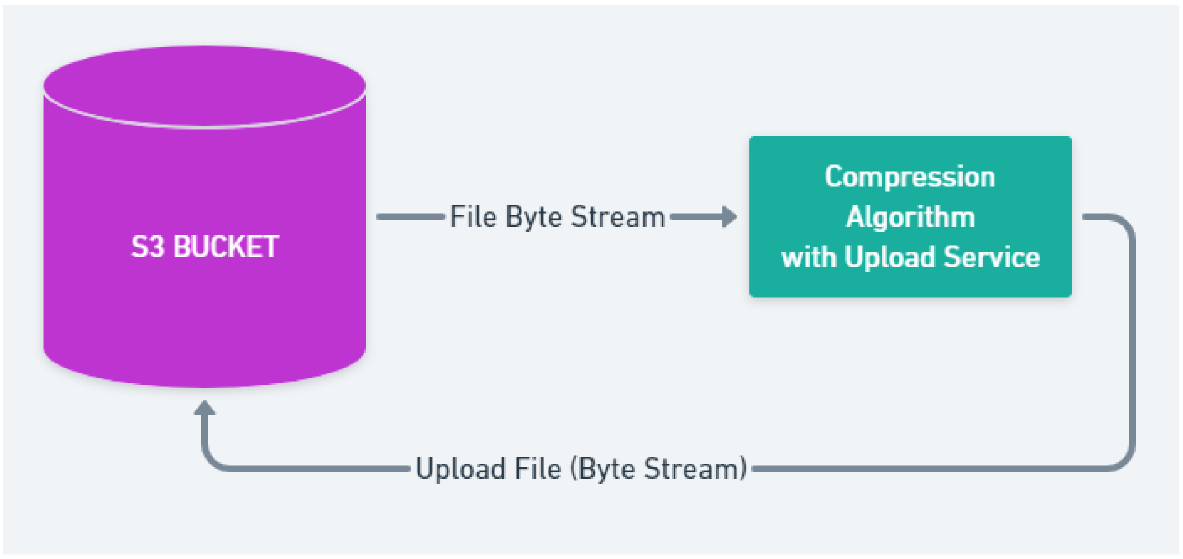

S3 BUCKET
File Byte Stream
Compression Algorithm with Upload Service
Upload File (Byte Stream)

Figure 2: File Compression Optimised Approach

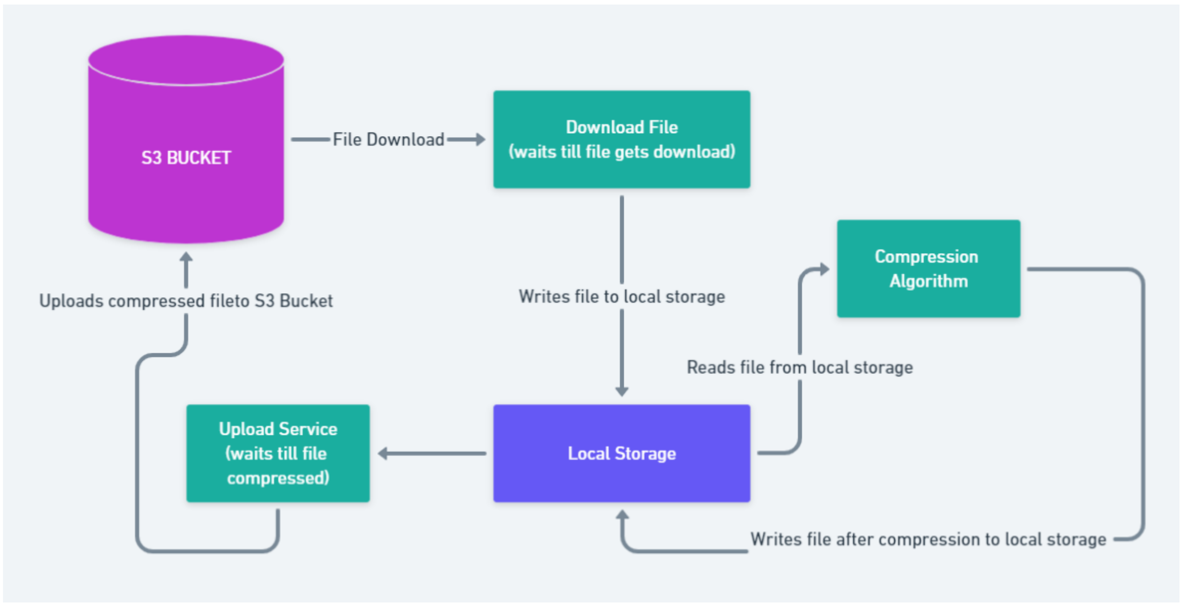

S3 BUCKET
File Download
Download File (waits till file gets download)
Writes file to local storage
Compression Algorithm
Reads file from local storage
Uploads compressed fileto S3 Bucket
Upload Service (waits till file compressed)
Local Storage
Writes file after compression to local storage

Figure 3: File Compression Default Approach

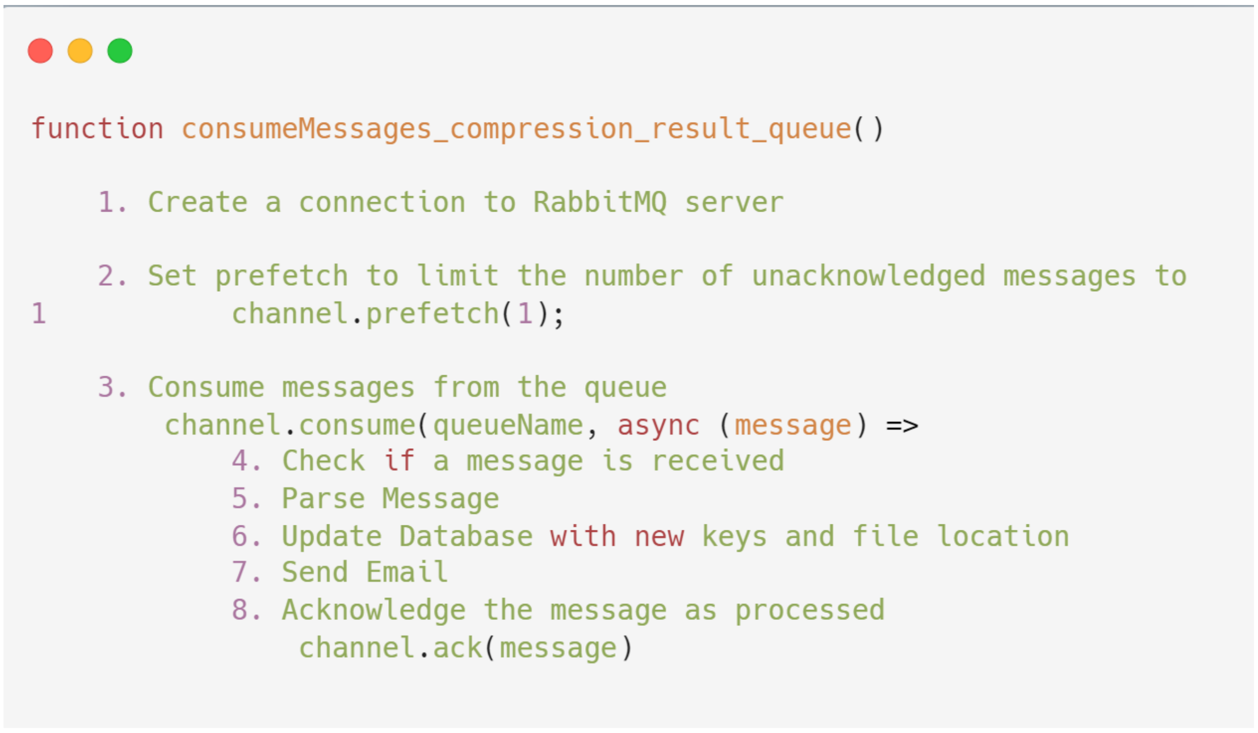
The conventional approach, in figure 3 for handling file uploads and storage involves waiting for the entire file to be uploaded and stored locally on the server. Once fully uploaded, the server reads the complete file from local storage into memory, compresses the file data using an algorithm like gzip or bzip2, writes the compressed data back to local storage, and finally uploads the compressed file to a remote storage service like Amazon S3. This method requires sufficient local storage space to temporarily hold both the original and compressed versions of potentially large files. Additionally, the multiple read/write operations to local storage can be slow and inefficient, and holding the entire file in memory during compression can strain the server's memory resources for very large files.

In contrast, the streaming byte data method (in figure 2) streamlines this process by eliminating the need for excessive local storage and reducing memory usage. As the file is being uploaded, the server reads the incoming byte stream directly from the network socket, without storing the entire file locally.

The byte stream is simultaneously compressed on-the-fly using a compression algorithm like gzip or bzip2, without storing the original file data in memory. The compressed byte stream is then immediately uploaded to the remote storage service without any intermediate local storage.

This approach significantly reduces the need for local storage space, as the original file is never stored locally. By avoiding storing the entire file in memory, it also reduces the memory footprint, making it more suitable for handling very large files. The compression and uploading happen concurrently as the file is being uploaded, resulting in faster processing times, especially for large files. It eliminates the multiple read/write operations to local storage, improving overall efficiency.

Upon completing the compression process, the results are dispatched to a designated result queue. The compressed file's key and URL are updated in the database, and a notification email is sent to the user with whom the file is shared.



```
function consumeMessages_compression_result_queue()

    1. Create a connection to RabbitMQ server

    2. Set prefetch to limit the number of unacknowledged messages to
            channel.prefetch(1);

    3. Consume messages from the queue
        channel.consume(queueName, async (message) =>
            4. Check if a message is received
            5. Parse Message
            6. Update Database with new keys and file location
            7. Send Email
            8. Acknowledge the message as processed
                channel.ack(message)
```

Figure 4: Result Queue Pseudo Code

The usage of RabbitMQ ensures that each request is eventually processed, even if the server restarts after a crash, ensuring dependability. Kubernetes is critical in dynamically expanding compression worker nodes to handle rising request volumes, allowing for quicker user request processing. Furthermore, it assists in scaling down nodes once all requests have been satisfied, optimising resource utilisation and lowering expenses.

RabbitMQ and Kubernetes work together to optimise load balancing and strengthen our system's stability. RabbitMQ orchestrates a decentralised queue system by implementing the Advanced Message Queuing Protocol (AMQP), in which compression requests are sent to a dedicated queue on a different server [22]. At the same time, Kubernetes manages the dynamic scaling of compression worker nodes in response to the changing length of the compression request queue.
As the demand for compression activities fluctuates, this dynamic placement of nodes guarantees an optimal distribution of processing resources. Furthermore, RabbitMQ's dedication to permanent message storage and delivery techniques ensures the eventual

execution of every request, even in the face of server restarts following a crash, reinforcing our system's stability.

Kubernetes, on the other hand, is critical in dynamically altering the number of compression worker nodes based on system demand, reducing resource fatigue and potential server failures. The orchestration platform also aids in fault recovery by restarting compression nodes automatically in the case of faults during request processing. RabbitMQ and Kubernetes' seamless collaboration not only improves load balancing but also optimises resource use, enabling a durable and responsive cloud data transmission architecture.

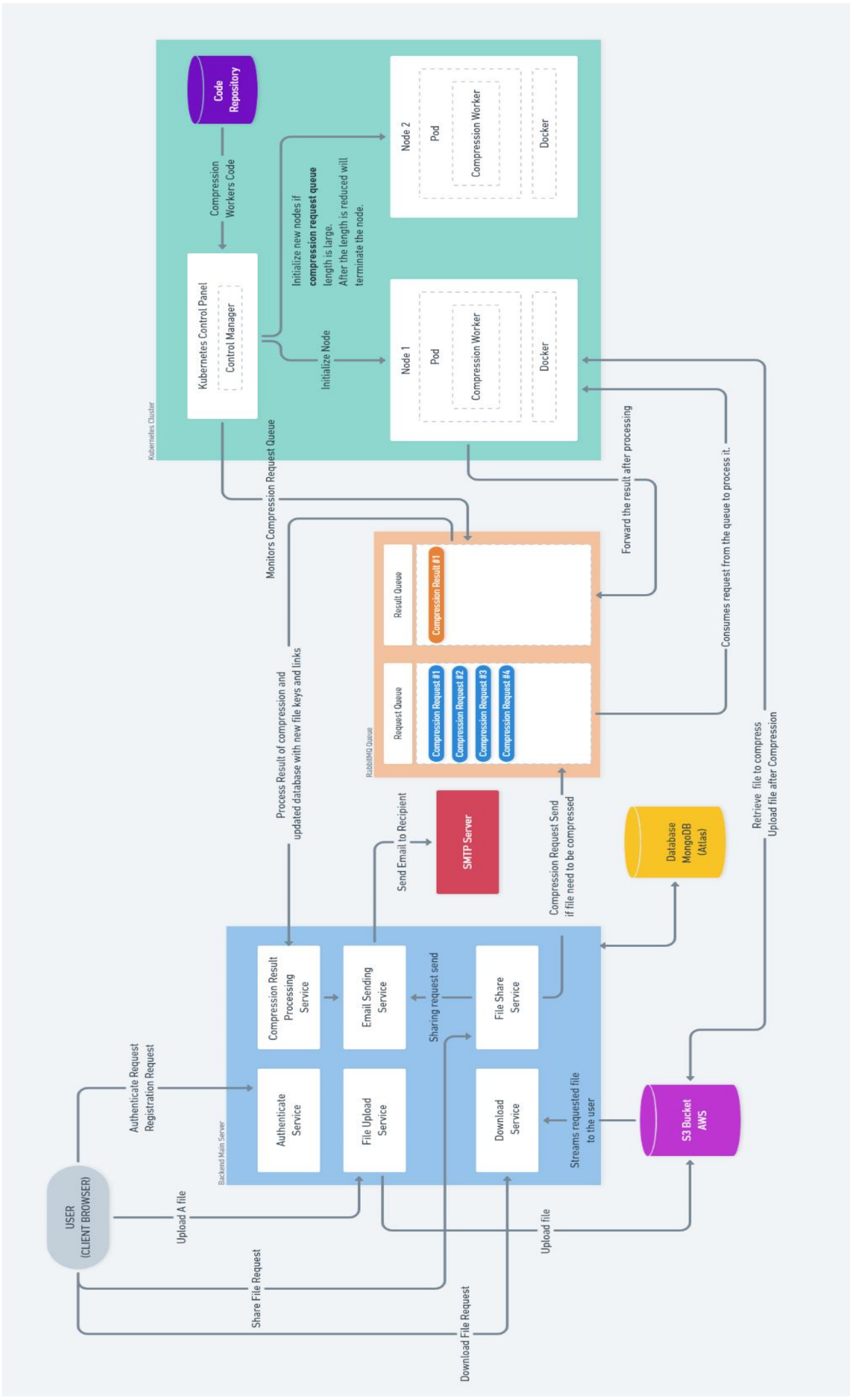


Figure 5: Architecture of our proposed solution with Kubernetes and RabbitMQ

Following Figure 5, the architecture of our proposed solution is a testament to the synergy between modern cloud infrastructure management and message queuing technologies. The diagram provides a visual representation of the intricate system designed to enhance the efficiency and reliability of cloud data transfer, particularly when dealing with large files that require compression.

Kubernetes Control Panel and Worker Nodes:

At the core of the architecture lies the Kubernetes Control Panel, which serves as the command center for orchestrating the deployment and management of containerized Compression Worker nodes. These nodes are dynamically provisioned across a cluster of servers, ensuring that the system can scale horizontally to meet varying demand levels. The Kubernetes Control Panel monitors the length of the compression request queue and adjusts the number of active worker nodes accordingly. This elasticity is crucial for maintaining high availability and performance without incurring unnecessary costs due to over-provisioning.

RabbitMQ Queuing System:

RabbitMQ plays a pivotal role in the architecture, acting as the message broker that decouples the compression tasks from the main application server. By routing compression requests to a dedicated queue, RabbitMQ ensures that the system can handle a high volume of concurrent requests without the risk of server crashes. This queuing system also provides resilience, as it can requeue tasks in the event of node failure, ensuring that no request is lost.

Amazon S3 Storage Integration:

The architecture seamlessly integrates with Amazon S3, a scalable cloud storage solution, to store and retrieve file data. The use of S3 allows the system to leverage AWS's robust infrastructure for data durability and accessibility. The streaming technique employed by the Compression Worker nodes to interact with S3 storage minimizes the need for local storage and reduces the memory footprint, which is particularly beneficial when processing large files.

Email Notification System:

An automated email notification system is incorporated into the architecture to inform users of the completion of compression tasks. This system is triggered once the compressed file is successfully uploaded back to S3 and the database is updated with the new file key and URL. The email notification provides users with a link to access the compressed file, enhancing the user experience by keeping them informed and engaged.

Database and Code Repository:

The architecture includes a database component, which is responsible for maintaining records of file metadata, compression results, and user information. This database is essential for tracking the state of each file throughout its lifecycle in the system. Additionally, a code repository is included to manage the versioning and deployment of the system's codebase, ensuring that updates and maintenance can be performed with minimal disruption.

In conclusion, the architecture depicted in Figure 5 is a robust and scalable solution for cloud data transfer and compression. It leverages the strengths of Kubernetes for container orchestration and RabbitMQ for message queuing, combined with the reliability of Amazon S3 storage. This design ensures that the system can handle large-scale compression tasks with high efficiency and reliability, providing a seamless experience for users and maintaining the integrity of their data.

3.3 IMPLEMENTATION

3.3.1 COMPRESSION WORKER

This Node.js script is intended to be a consumer of a RabbitMQ message queue, especially the "compression_request_queue." Its major goal is to process incoming messages by extracting information, getting a corresponding video file from AWS S3, compressing the video with FFmpeg, and then sending the compressed video back to S3.

The AWS S3 setup is set up with the AWS SDK and credentials obtained via environment variables. Similarly, the RabbitMQ connection URL is retrieved from the environment variables. To manage promises efficiently, the script makes use of asynchronous functions and the 'await' keyword. Temporary files are created with the 'tmp' library and are deleted after the compression operation is finished.

Following a successful compression and upload, the script sends a message to a new RabbitMQ queue called "compression_result_queue," which contains information about the compressed movie. To provide resilience, the entire process is encased behind a thorough error-handling structure. In summary, this script connects RabbitMQ with AWS S3, resulting in a video compression service that listens for requests, compresses them, and sends the results via a separate message queue.

```
def consume_messages():
    try:
        queue_name = "compression_request_queue"
        connection = amqp.connect(rabbitmqURL)
        channel = connection.create_channel()
        channel.queue_declare(queue=queue_name, durable=False)
        print(f"Waiting for messages from queue \"{queue_name}\"...")

        def callback(ch, method, properties, body):
            json_message = json.loads(body)
            print(f"Received message from queue \"{queue_name}\":", json_message)

            s3_key = json_message["s3Key"]
            download_params = {"Key": s3_key, "Bucket": os.getenv("AWS_S3_BUCKET")}
            download_stream = s3.get_object(**download_params)["Body"].read()

            tmp_file = tmp.NamedTemporaryFile(delete=False, 
suffix=os.path.splitext(s3_key)[1])
            output_file_path = tmp_file.name

            ffmpeg.input("pipe:").output(output_file_path).run(input_data=download_stream)

            upload_params = {"Key": f"compressed_{s3_key}", "Body": 
open(output_file_path, "rb"), "Bucket": os.getenv("AWS_S3_BUCKET")}
            upload_result = s3.upload_file(**upload_params)

            print(upload_result)
            send_to_compression_result_queue({
                "CompressedFileUrl": upload_result["Location"],
                "CompressedFilekey": upload_result["Key"],
                "shareid": json_message["shareid"]
            })
            print("Compression Done")
            ch.basic_ack(delivery_tag=method.delivery_tag)

        channel.basic_consume(queue=queue_name, on_message_callback=callback)
        channel.start_consuming()

    except Exception as error:
        print("Error:", error)
```

3.3.2 COMPRESSION RESULT PROCESSING WORKER CODE

The provided code defines a Node.js function intended for consuming messages from a RabbitMQ queue named "compression_result_queue." The function establishes a connection to the RabbitMQ server using the AMQP library and processes incoming messages asynchronously. Upon receiving a message, the function parses it as JSON and updates a document in a MongoDB collection using Mongoose. Specifically, it sets properties related to the compression status and the location of the compressed file. Subsequently, the function triggers an email notification using the email-handler utility, providing details such as the sender's name, the file's download URL, and the email address of the recipient. Finally, the function acknowledges the receipt of the message to RabbitMQ.

In summary, this code segment is responsible for handling messages from the "compression_result_queue," updating a MongoDB document with compression-related information, sending an email notification, and acknowledging the message receipt.

```
const amqp = require("amqplib");
const fileShareModel = require("../models/fileShareModel");
const emailHandler = require("../utils/email-handler.js");
const rabbitmqURL = process.env.AMQP_URL;

async function consumeMessages_compression_result_queue() {
  try {
    const queueName = "compression_result_queue";
    const connection = await amqp.connect(rabbitmqURL);
    const channel = await connection.createChannel();
    channel.prefetch(1);
    await channel.assertQueue(queueName, { durable: false });
    console.log(`Waiting for messages from queue "${queueName}"...`);

    channel.consume(queueName, async (message) => {
      if (message !== null) {
        const jsonMessage = JSON.parse(message.content.toString());
        console.log(`Received message from queue "${queueName}":`, jsonMessage);

        const uploadResult = await fileShareModel.findOneAndUpdate(
          { shareid: jsonMessage.shareid },
          {
            "file.compressed": true,
            "file.location": jsonMessage.CompressedFileUrl,
            "file.key": jsonMessage.CompressedFilekey,
          },
          { new: true }
        );
        console.log(jsonMessage.shareid);
        emailHandler.sendFileSharingEmail({
          senderName: uploadResult.file.shared_by,
          fileUrl: process.env.BaseUrl + "/download/" + uploadResult.shareid,
          emailReceiver: "<" + uploadResult.email + ">",
        });
        console.log("Email sent to: ", uploadResult.email);
        channel.ack(message);
      }
    });
  } catch (error) {
    console.error("Error:", error);
  }
}
module.exports = consumeMessages_compression_result_queue;
```

3.4 WEB INTERFACE

Our user-centric web interface acts as the user-centric gateway to our secure cloud data transfer system, guaranteeing a smooth and straightforward experience throughout the data transfer lifecycle.

3.4.1 AUTHENTICATION AND REGISTRATION

The registration and authentication module ensures that users have a safe onboarding experience. Users may easily register accounts using a simplified registration interface, and sophisticated authentication procedures protect user data and provide secure access to the system.

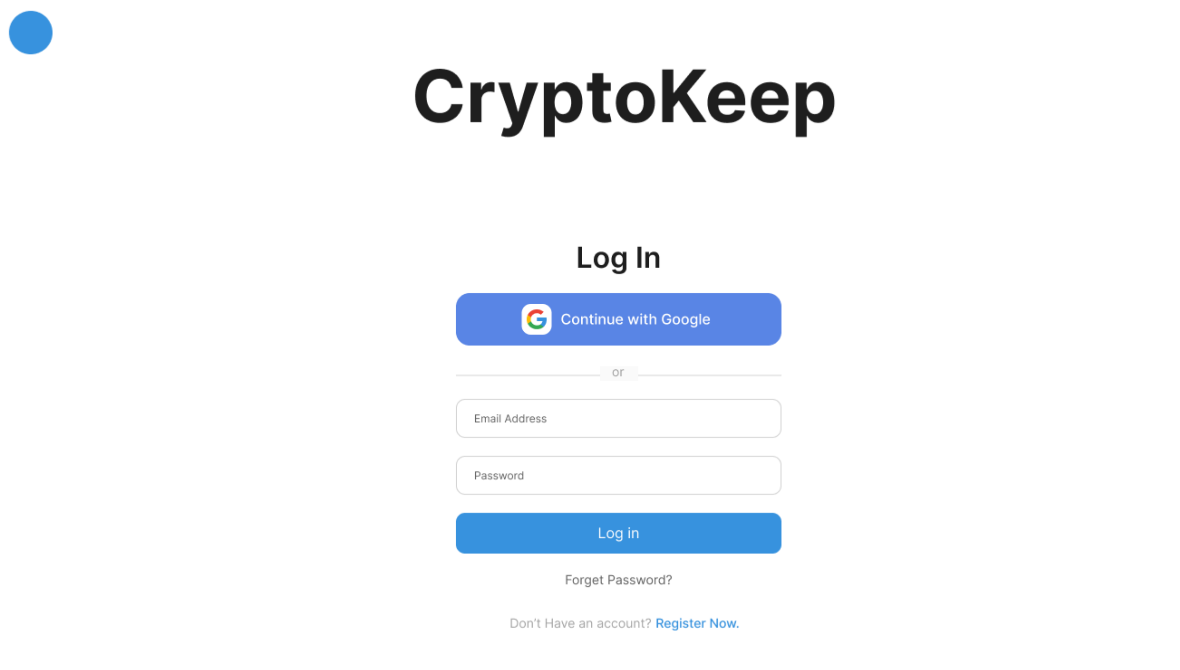


Figure 6: Login Page Interface

Figure 6 showcases the login page interface, designed with simplicity and security in mind. Upon accessing our platform, users are greeted with a clean and intuitive login page. Here, users can securely log in to their accounts by entering their credentials, which are verified through robust authentication procedures before granting access to the system.

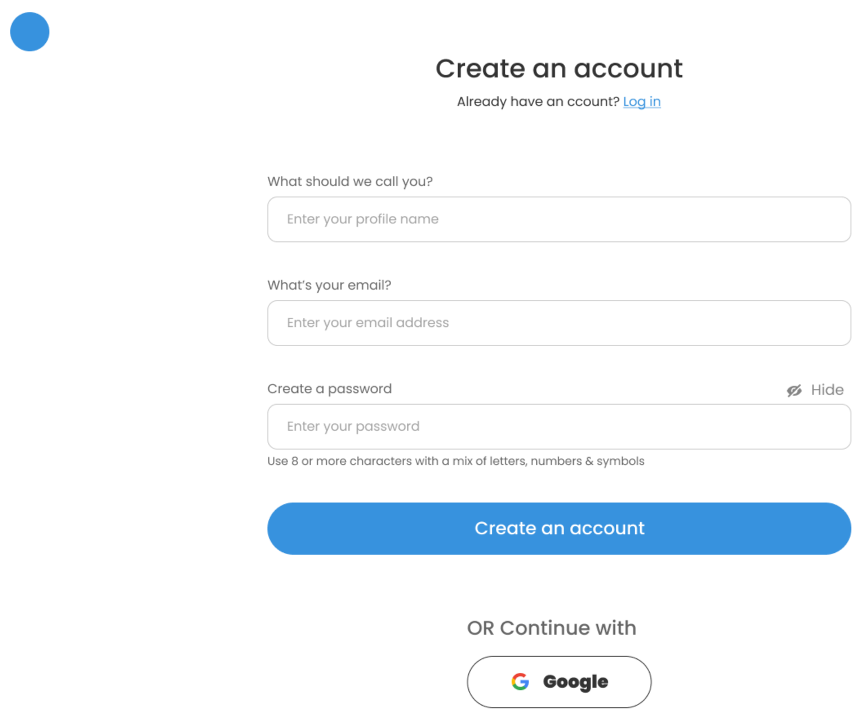


Figure 7: Signup Page Interface

Figure 7 presents the signup page interface, providing users with a straightforward registration process. Our signup page is designed to streamline the account creation process, guiding users through a series of minimalistic and easy-to-follow steps. Users are prompted to input necessary registration details, such as username, email address, and password, ensuring a seamless registration experience. Behind the scenes, advanced authentication protocols safeguard user data, including encryption techniques and secure password hashing, to fortify the integrity of user accounts and protect against unauthorized access.

By leveraging these intuitive registration and authentication interfaces, we prioritize both user convenience and data security, fostering trust and confidence in our platform's user experience.

3.4.2 UPLOAD INTERFACE

The upload interface is intended to be simple and efficient. Users may easily upload files by utilising an easy and user-friendly interface. This module supports a wide range of file types and sizes, allowing users to easily begin the data transfer process.

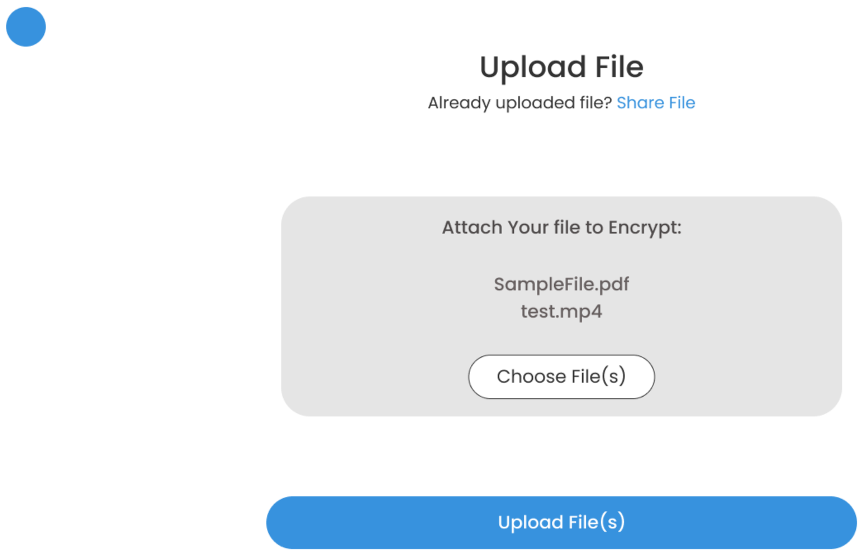
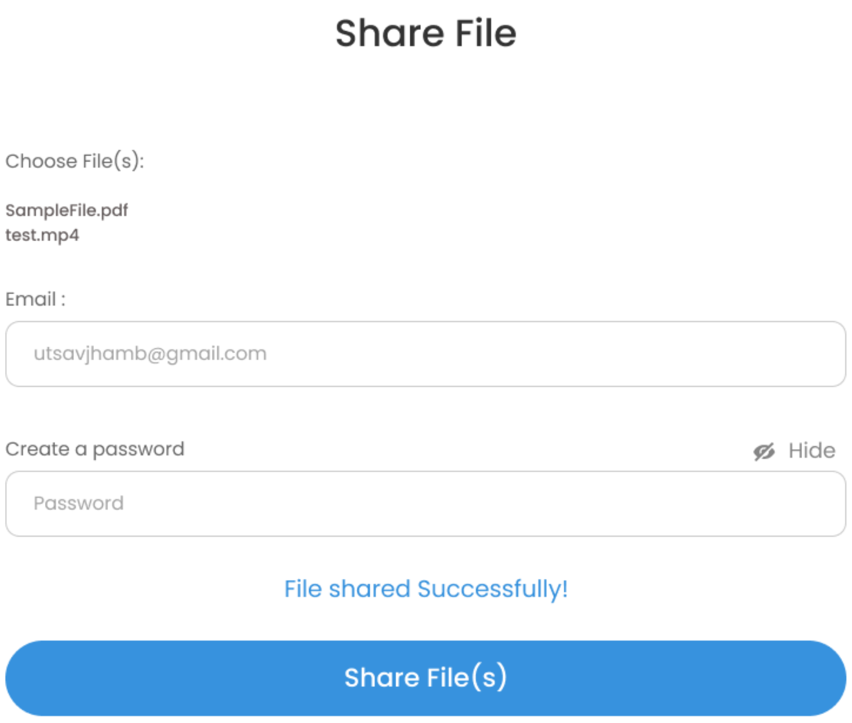

Upload File
Already uploaded file? Share File
Attach Your file to Encrypt:
SampleFile.pdf
test.mp4
Choose File(s)
Upload File(s)

Figure 8: Upload File Interface

3.4.3 SHARE INTERFACE(S)

Sharing data securely is a key feature made possible by our online interface. The share interface allows users to designate recipients and define access rights, allowing for regulated and secure file sharing. This feature improves user collaboration and information distribution.


Share File

Choose File(s):

SampleFile.pdf
test.mp4

Email :

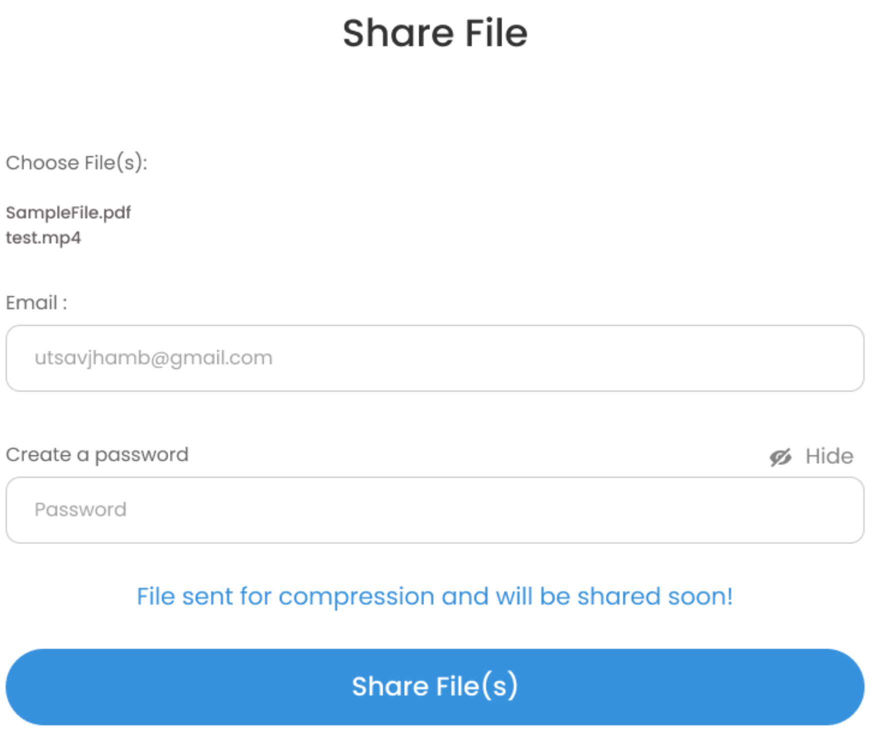
utsavjhamb@gmail.com

Create a password

Password

File shared Successfully!

Figure 9: Share File Without Compression Interface


Share File

Choose File(s):

SampleFile.pdf
test.mp4

Email :

utsavjhamb@gmail.com

Create a password

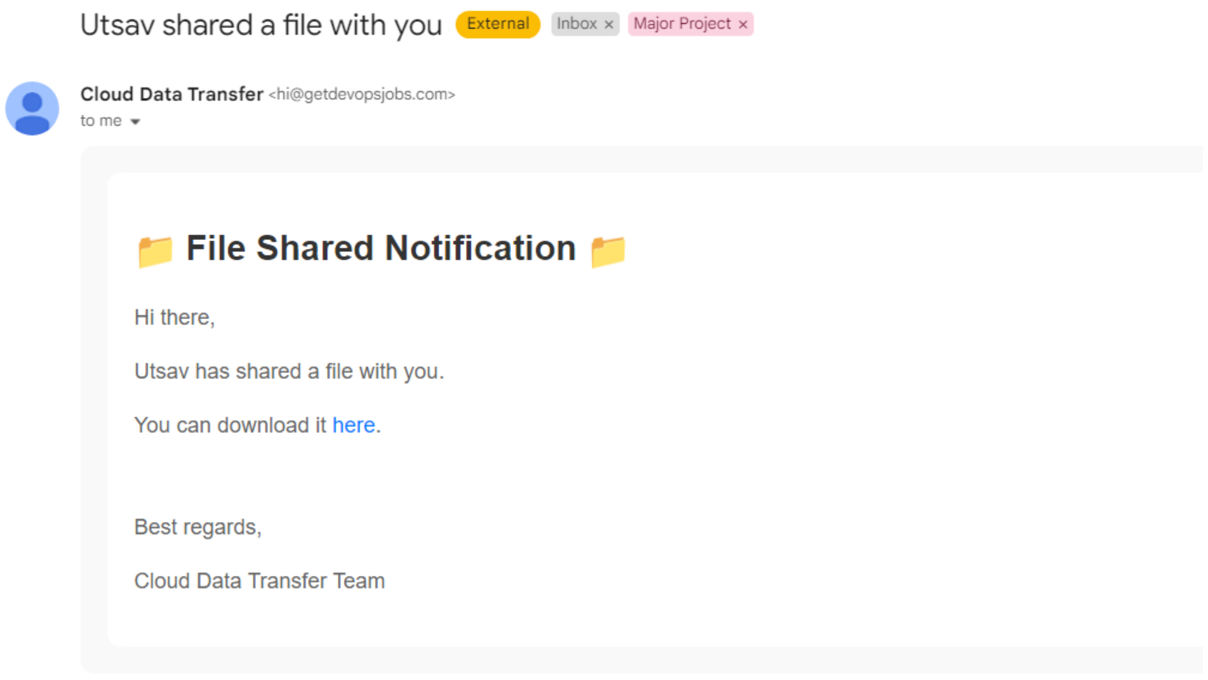
Password

File sent for compression and will be shared soon!

Figure 10: Share File With Compression Interface

3.4.4 EMAIL NOTIFICATION

Our system relies on keeping users informed. The email notification module delivers automatic emails to users when compression procedures are completed successfully. This not only keeps consumers informed, but also allows them to track and regulate their data flows.


Utsav shared a file with you External Inbox × Major Project ×

Cloud Data Transfer <hi@getdevopsjobs.com>
to me

📁 File Shared Notification 📁

Hi there,

Utsav has shared a file with you.

You can download it here.

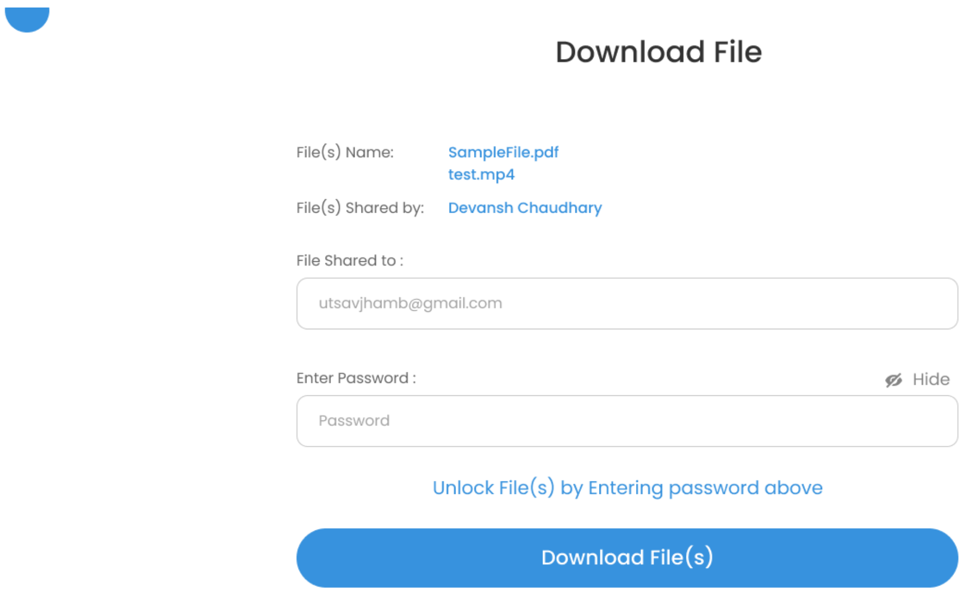
Best regards,

Cloud Data Transfer Team

Figure 11: Email Notification Screenshot

3.4.5 DOWNLOAD INTERFACE

The download interface facilitates the retrieval of compressed files with simplicity and speed. Users can access their shared files securely through a dedicated download interface, completing the data transfer cycle seamlessly.


Download File

File(s) Name: SampleFile.pdf
test.mp4

File(s) Shared by: Devansh Chaudhary

File Shared to : utsavjhamb@gmail.com

Enter Password : Hide

Password

Unlock File(s) by Entering password above

Download File(s)

Figure 12: Download File Interface

3.4.6 SYSTEM INTERFACE FLOWCHART

To provide a comprehensive understanding of the user journey within our web interface, we have included a detailed flowchart that visually represents the interaction between the various modules and interfaces of our system.

Figure 13 illustrates the flow of actions a user can take, from logging in or signing up for an account to uploading, sharing, and downloading files. This flowchart serves as a visual guide to the logical sequence of operations and decision points that users encounter as they navigate through our secure cloud data transfer system.

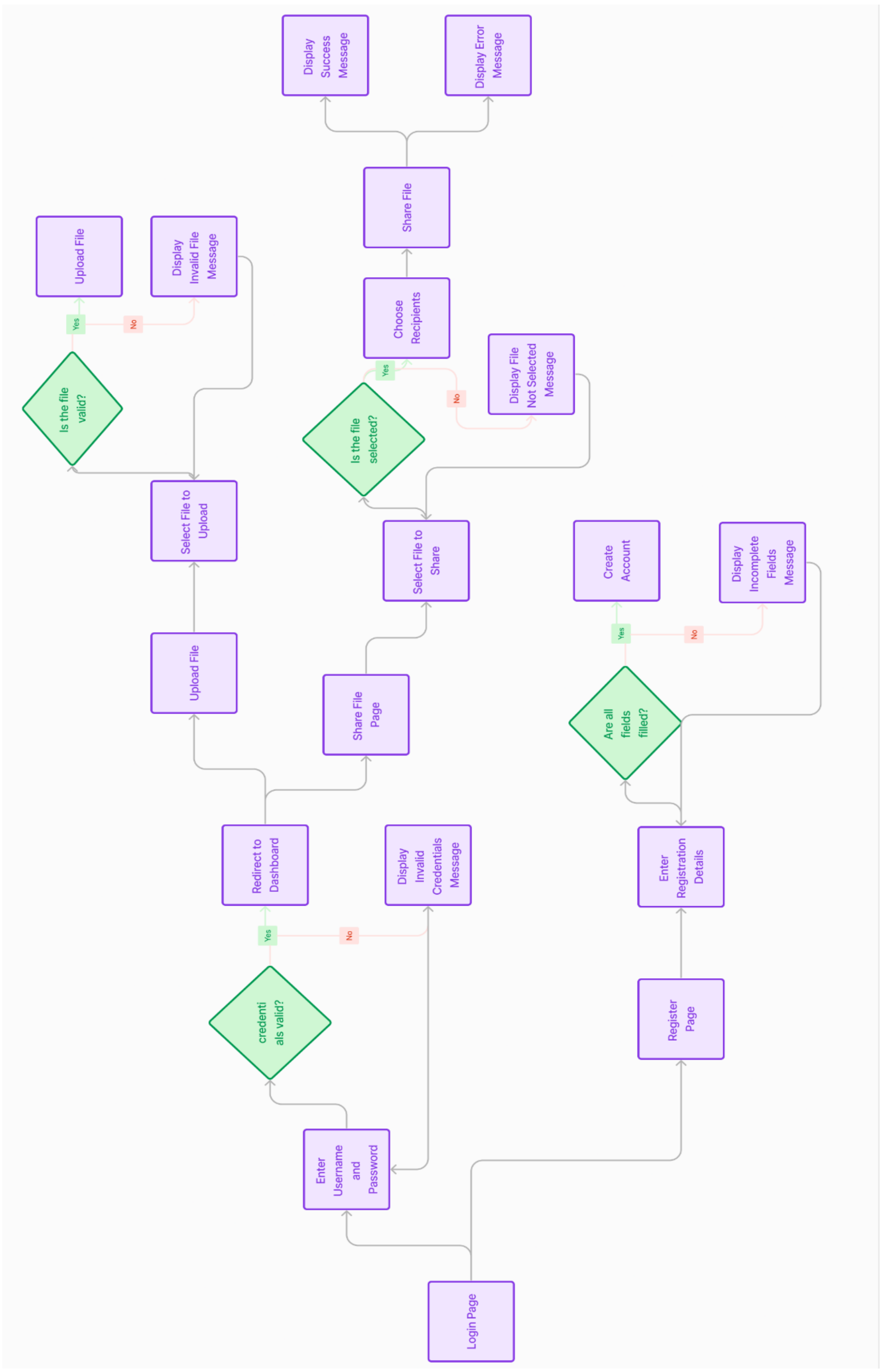


Figure 13: System Interface Flowchart

3.5 KEY CHALLENGES

a. Managing Large Tasks Without Crashing

The Problem: When dealing with large amounts of data, our servers might get overburdened and crash. This is equivalent to carrying too much and dropping everything. How We Dealt With It: Kubernetes, our virtual manager, comes into play. It's like having a super-smart organiser who calls in reinforcements (compression worker nodes) when the job becomes too much. These worker nodes handle the hard labour and elegantly quit when finished. This dynamic scaling aids in the prevention of server crashes.

b. Making Compression Dependable and Versatile

The Problem: Some data compression technologies are ineffective when dealing with large numbers of users or complex jobs. It's like trying to cram too much into a tight box - things might get out of hand.

How We Dealt With It: Kubernetes takes another step forward. It monitors the compression request queue and modifies the number of compression worker nodes based on the amount of jobs that are waiting. As a result, our system remains adaptable and dependable, smoothly handling shifting workloads. RabbitMQ also helps to keep things organised by ensuring that compression requests are fulfilled in a timely way.

c. Avoiding Chaos with a Large Number of Users

The Problem: Imagine everyone trying to compress their files at the same time; this might cause pandemonium and perhaps destroy our system.

How We Dealt With It: We utilise RabbitMQ to construct a queue or queue for compression requests rather than permitting a free-for-all. Each request waits patiently for its time, avoiding a chaotic rush. This not only maintains order but also decreases the danger of server breakdowns during periods of high user activity.

d. Even after a crash, data can be saved.

The Problem: Server breakdowns are like unforeseen obstacles, and if not managed appropriately, they can result in data loss.

How We Dealt With It: RabbitMQ demonstrates its tenacity once more. It maintains track of all compression requests that are queued. So, even if our server crashes and has to restart, it resumes where it left off, guaranteeing that no data is lost. Furthermore, if a compression worker node meets a problem, the request is simply requeued, and the node is given another opportunity.

e. Streaming allows you to complete tasks more quickly.

The Difficulty: Traditional techniques entail a step-by-step procedure of obtaining files, storing them, compressing them, and finally uploading. It takes time and space to make a sandwich, wrap it, and then consume it.

How We Dealt With It: We used a streaming strategy, which is similar to constructing a sandwich on the go. We process huge files in real-time instead of downloading and storing them. This not only expedites the process but also reduces the need for large amounts of server storage space, making it more efficient and agile.

CHAPTER 4: TESTING

4.1 TESTING STRATEGY

In the development of our secure cloud data transfer system, a comprehensive testing strategy has been devised to ensure the reliability and functionality of the entire system. This strategy encompasses multiple levels of testing, each designed to address specific aspects of the system's performance and security.

At the foundational level, unit testing focuses on verifying the functionality of individual components, such as compression nodes, RabbitMQ integration, and Kubernetes scaling mechanisms. This ensures that each component operates as intended in isolation. Moving to integration testing, we assess the seamless interaction between different system components, evaluating communication channels and ensuring integrated components work together without errors.

System testing, a pivotal phase, involves end-to-end tests that simulate real-world user scenarios. This includes assessing compression requests, dynamic scaling, and database updates. Furthermore, performance testing evaluates the system's responsiveness and resource utilization under various conditions, such as high loads and concurrent user requests. Security testing, a critical aspect, aims to identify and address potential vulnerabilities in the data transfer process, including penetration testing and access control assessments.

Scalability testing explores the system's ability to adapt to varying workloads by simulating increasing compression requests and observing how Kubernetes scales compression worker nodes. Lastly, reliability and recovery testing intentionally induces failures and crashes to observe how the system recovers, ensuring data consistency even in unexpected scenarios.

4.1.1 LOCUST TESTING

As a pivotal component of our comprehensive testing strategy, Locust testing and stress testing play distinctive roles in ensuring the robustness and reliability of our secure cloud data transfer system.

Locust testing is a targeted approach aimed at examining specific focal points or critical areas within the system. In the context of our secure cloud data transfer, Locust testing involves a meticulous examination of key functionalities, components, or modules. This focused testing strategy allows us to ensure that critical aspects, such as the compression algorithm, RabbitMQ integration, and Kubernetes dynamic scaling, perform optimally under varying conditions [24].

For example, in Locust testing of the compression algorithm, we scrutinize its efficiency, accuracy, and ability to handle diverse file types. Similarly, Locust testing of RabbitMQ assesses its message queuing capabilities, ensuring that compression requests are appropriately queued and processed in the desired order. Locust testing aids in identifying and rectifying issues specific to crucial components, contributing to the overall reliability of the system [25].

4.1.2 STRESS TESTING

Stress testing, on the other hand, is designed to evaluate the system's robustness under extreme conditions and beyond its anticipated operational capacity. This type of testing simulates scenarios where the system is subjected to high loads, concurrent user requests, or resource constraints to assess its performance boundaries and potential points of failure.

In the context of our secure cloud data transfer system, stress testing involves pushing the system to its limits in terms of compression requests, file sizes, and concurrent user interactions. By doing so, we aim to identify how the system behaves under intense pressure, uncover potential bottlenecks, and ensure that it gracefully degrades rather than failing catastrophically. Stress testing is crucial for understanding the system's scalability, resource utilization, and its ability to recover from adverse conditions.

In summary, while Locust testing hones in on specific critical areas, stress testing takes a more holistic approach by subjecting the entire system to extreme conditions. Together, these testing methodologies contribute to a robust testing strategy that ensures the resilience and reliability of our secure cloud data transfer system across various scenarios and usage patterns.

4.2 TEST CASES AND OUTCOMES

Within these testing categories, specific test cases have been developed to assess critical functionalities. In the compression process category, test cases focus on validating the accuracy of file compression and ensuring that compressed data aligns with expectations. Queue management test cases simulate different scenarios of compression requests in the queue, evaluating how RabbitMQ handles queuing to maintain the correct order of processing.

Dynamic scaling scenarios are tested to trigger varying compression request loads and observe Kubernetes' response, confirming its ability to dynamically scale compression worker nodes based on workload changes. Additionally, data integrity test cases assess the accuracy and consistency of data updates in the database after the completion of the compression process.

This comprehensive set of test cases ensures that the system not only meets functional requirements but also performs reliably, securely, and efficiently under diverse conditions. The outcomes of these tests will guide us in refining and optimizing the system to deliver a secure and robust cloud data transfer solution.

CHAPTER 5: RESULTS AND EVALUATION

5.1 RESULTS

In our study, we conducted a comprehensive comparison of three deployment scenarios for the cloud data transfer and compression system: a single instance setup, deployment with Kubernetes autoscaling with max allowed nodes set as 5, and deployment with Kubernetes scaling based on compression queue length in RabbitMQ with max allowed nodes set as 5 and scale at multiple of 5 requests.

Single Instance Deployment:

The single instance deployment served as the baseline for our comparison. In this configuration, a lone server handled compression requests without dynamic scaling capabilities. The results indicated limitations in handling concurrent requests, leading to potential bottlenecks during periods of high demand. The lack of scalability resulted in longer processing times and an increased risk of server crashes, especially under heavy workloads.

Kubernetes Autoscaling Deployment:

The Kubernetes autoscaling deployment demonstrated significant improvements in performance and reliability. By leveraging Kubernetes, the system dynamically allocated and unallocated compression worker nodes based on overall demand. The autoscaling mechanism effectively managed varying compression queue lengths, ensuring optimal resource utilization. This approach resulted in faster request processing times, improved system responsiveness, and a reduced likelihood of server crashes under varying workloads.

Kubernetes Scaling based on Compression Queue Length:

Our innovative approach involved dynamically scaling compression worker nodes based on the length of the compression request queue in RabbitMQ. As the queue length increased, new nodes were deployed to handle the surge in requests, and surplus nodes were scaled down during periods of low demand. This strategy exhibited superior adaptability to fluctuations in workload, offering an efficient balance between resource utilization and responsiveness. The dynamic scaling based on compression queue length proved effective in preventing potential bottlenecks and optimizing overall system performance.

For the same compute power resources, we will be taking 5 single instances and a kubernetes cluster with scaling between 0 to 5 nodes. In single instances we can't scale down when we don't require and hence we get a fixed cost that is usually high. Whereas in kubernetes with auto scaling we can switch the numbers of nodes running according to the processing demand required to write now. We have assumed here that all requests are sent in batches at the start of the hour.

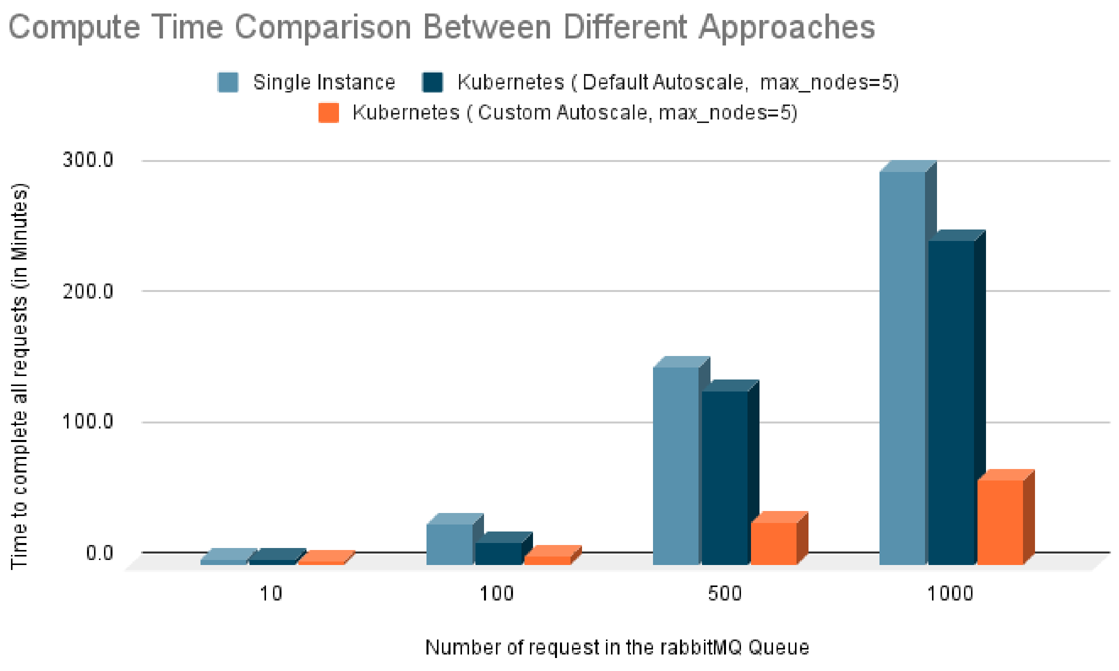


Figure 14: Compute Time Comparison Between Different Approaches Graph

No of requests send in	Single Instance	Kubernetes (Default Autoscale, max_pods=5)	Kubernetes (Custom Autoscale, max_pods=5)
10	3.0	3.0	1.8
100	30.0	16.7	6.4
500	150.0	132.5	32.2
1000	300.0	246.7	64.2

Table 2: Compute Time Comparison Between Single Instance, Kubernetes Autoscaling, and Kubernetes Scaling based on Queue Length

The comparison highlights the substantial benefits of leveraging Kubernetes for dynamic scaling in cloud data transfer and compression systems. While Kubernetes autoscaling provides an effective solution, our innovative approach of scaling based on compression queue length in RabbitMQ offers additional advantages in terms of adaptability and resource optimization.

The choice between these approaches depends on specific performance requirements, workload characteristics, and cost considerations. Ultimately, our methodology provides a flexible framework that can be tailored to meet diverse operational needs in cloud-based data transfer and compression scenarios.

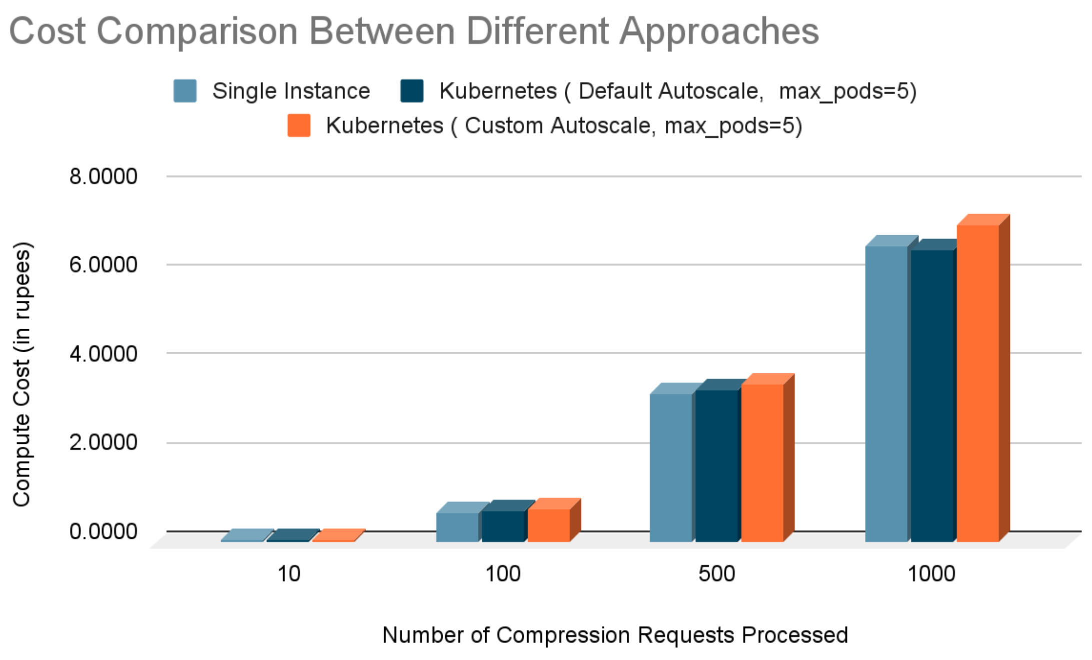


Figure 15: Absolute Cost Comparison Between Different Approaches Graph

No of Compression request	**Single Instance**	**Kubernetes (Default Autoscale, max_pods=5)**	**Kubernetes (Custom Autoscale, max_pods=5)**
10	0.0667	0.0800	0.0815
100	0.6667	0.7144	0.7430
500	3.3333	3.4333	3.5741
1000	6.6667	6.5778	7.1296

Table 3: Absolute Cost Comparison Between Single Instance, Kubernetes Autoscaling, and Kubernetes Scaling based on Queue Length Graph

The table 3 compares the absolute cost between three different configurations: a single instance, Kubernetes with default autoscaling with a maximum of 5 pods, and Kubernetes with custom auto scaling also with a maximum of 5 pods. For a workload of 10 units, the single instance had the lowest absolute cost at 0.0667, followed by the default Kubernetes autoscaling at 0.0800, and the custom Kubernetes autoscaling was highest at 0.0815.

As the workload increased to 100 units, the single instance cost was 0.6667, the default auto scaling was 0.7144, and the custom scaling was 0.7430. At 500 units, the single instance was 3.3333, default scaling 3.4333, and custom scaling the highest at 3.5741. However, when the workload reached 1000 units, the custom Kubernetes autoscaling configuration had the lowest absolute cost at 7.1296, compared to 6.6667 for the single instance and 6.5778 for the default autoscaling.

This suggests that for higher workloads, the custom Kubernetes autoscaling configuration becomes more cost-effective than a single instance or the default auto scaling settings. The ability to customize the scaling allowed better utilization of resources as demands increased. But for lighter workloads up to 500 units, the single instance was the most cost-efficient option based on this data with a maximum of 5 pods.

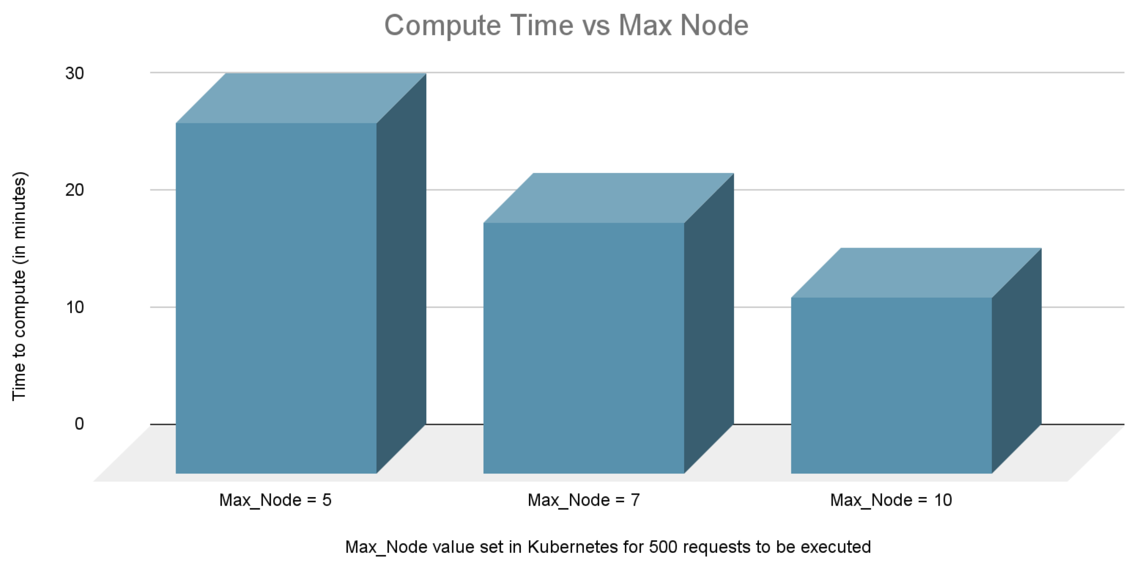


Figure 16: Compute Time vs Max Node Graph

Max_Node value set as	**Time to Compute (in minutes)**
Max_Node = 5	30
Max_Node = 7	21.5
Max_Node = 10	15

Table 4: Compute Time vs Max Node Graph

The table 4 presents data that compares the compute time required for different maximum node configurations in a system or environment. Specifically, it showcases the compute time when the maximum number of nodes is set to 5, 7, and 10, respectively.

From the data, it is evident that increasing the maximum number of nodes results in a reduction in the compute time required to complete a task or process. When the maximum number of nodes is set to 5, the compute time is 30 units. However, when the maximum number of nodes is increased to 7, the compute time decreases to 21.5 units, reflecting a significant improvement in performance.

Further increasing the maximum number of nodes to 10 leads to an even greater reduction in compute time, with the value dropping to 15 units. This trend suggests that allocating more nodes to the system can effectively distribute the workload across multiple resources, leading to faster computation and improved overall performance.

It is important to note that while increasing the maximum number of nodes can enhance the compute time, there may be trade-offs in terms of resource utilization, cost, and system complexity. Determining the optimal number of nodes often involves balancing these factors based on the specific requirements and constraints of the application or environment.

Additionally, it would be beneficial to have more data points or a broader range of maximum node values to better understand the relationship between the number of nodes and compute time. This could provide insights into potential diminishing returns or other non-linear behaviors that may arise as the number of nodes increases further.

5.2 COMPARISON WITH EXISTING SOLUTIONS

The table compares three approaches to Kubernetes scaling: Single Instance, Kubernetes Autoscaling, and Kubernetes Scaling based on Queue Length, across key metrics such as scalability, reliability, processing time, and cost.

In terms of scalability, the Single Instance approach is limited, while Kubernetes Autoscaling shows improved scalability, and Kubernetes Scaling based on Queue Length demonstrates optimized scalability.

Regarding reliability, the Single Instance method poses a higher risk of server crashes, particularly during peak loads. In contrast, Kubernetes Autoscaling enhances reliability through dynamic node management, and Kubernetes Scaling based on Queue Length achieves improved reliability by proactively scaling based on workload.

In processing time, the Single Instance has longer processing times, while Kubernetes Autoscaling reduces processing times through dynamic scaling. Kubernetes Scaling based on Queue Length goes further by achieving additional reductions in processing times with adaptive scaling.

Lastly, the cost aspect indicates that the Single Instance approach results in suboptimal resource usage and increased costs. Kubernetes Autoscaling enables efficient resource utilization, making it cost-effective. Meanwhile, Kubernetes Scaling based on Queue Length optimizes resource usage and is associated with cost-efficient scaling strategies.

	Single Instance	**Kubernetes Autoscaling**	**Kubernetes Scaling based on Queue Length**
Scalability	Limited	Improved scalability	Optimized scalability
Reliability	Higher risk of server crashes, especially during peak loads	Enhanced reliability through dynamic node management	Improved reliability, proactive scaling based on workload
Processing Time	Longer processing times	Reduced processing times due to dynamic scaling	Further reduction in processing times with adaptive scaling
Cost	Suboptimal resource usage, increased costs	Efficient resource utilization, cost-effective	Optimized resource usage, cost-efficient scaling

Table 5: Comparing Single Instance, Kubernetes Autoscaling, and Kubernetes Scaling based on Queue Length

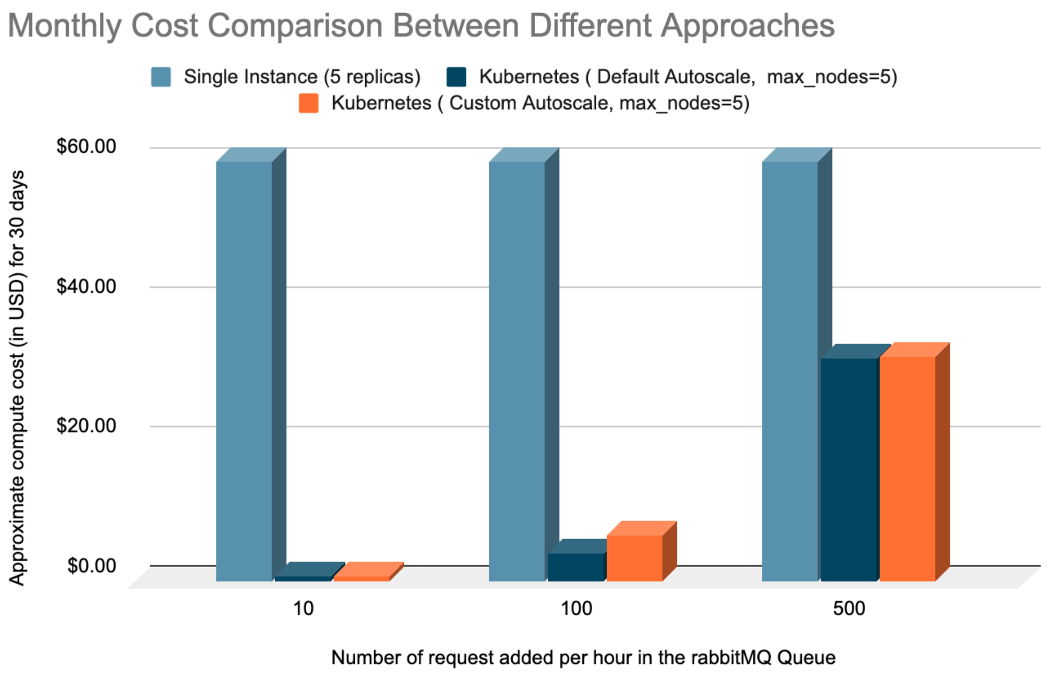


Figure 17: Monthly Cost Comparison Between Different Approaches Graph

Number of requests processed per hour	Monthly Cost (in USD)		
	Single Instance (5 replicas)	Kubernetes (Default Autoscale, max_pods=5)	Kubernetes (Custom Autoscale, max_pods=5)
10	60.000	0.720	0.733
100	60.000	4.000	6.417
500	60.000	31.800	32.167

Table 6: Monthly Cost Comparing Single Instance, Kubernetes Autoscaling, and Kubernetes Scaling based on Queue Length

The comparison of deployment strategies in table 6 and figure 16, reveals the cost advantages of leveraging Kubernetes autoscaling, especially when dealing with high traffic volumes or fluctuating demand. As the number of requests per hour increases, the cost benefits of Kubernetes autoscaling become more pronounced compared to a single instance deployment. For lower request rates, the cost differences between the deployment strategies are relatively minor, with the single instance being the most expensive option, followed by the default

Kubernetes autoscaling and the custom Kubernetes autoscaling with a maximum of 5 pods being slightly higher than the default autoscaling.

However, as the request rate rises, the cost advantage of Kubernetes autoscaling becomes apparent. The single instance cost remains fixed, while the default Kubernetes autoscaling and the custom Kubernetes autoscaling with a maximum of 5 pods exhibit potential cost savings compared to the single instance deployment.

The cost savings of Kubernetes autoscaling are most significant at higher request rates. In these scenarios, the single instance cost remains the highest, but the default Kubernetes autoscaling cost and the custom Kubernetes autoscaling with a maximum of 5 pods cost are substantially lower, representing substantial cost reductions compared to the single instance deployment.

The data demonstrates that Kubernetes autoscaling can optimize resource utilization and reduce operational costs by dynamically adjusting the number of pods based on the load. This makes Kubernetes autoscaling an attractive solution for enterprises seeking to maximize resource efficiency, minimize infrastructure expenses, and maintain high availability and scalability for their applications, particularly in scenarios with high traffic volumes or fluctuating demand.

Chapter 6: CONCLUSIONS AND FUTURE SCOPE

6.1 CONCLUSION

In this comprehensive exploration of secure cloud data transfer, our methodology, anchored by the symbiotic relationship between RabbitMQ's Advanced Message Queuing Protocol (AMQP) and Kubernetes, has demonstrated noteworthy achievements in addressing fundamental challenges. The strategic allocation of compression worker nodes dynamically based on queue length has proven to be a resilient solution, ensuring not only efficient resource utilization but also the scalability and reliability necessary for handling diverse workloads.

The adoption of a streaming approach for file byte data processing from S3 marks a paradigm shift in traditional methods. By circumventing the need for intermediate steps such as local saving, this approach significantly diminishes processing time and minimizes the demand for extensive server storage space. This innovation proves particularly advantageous when dealing with large file volumes, enhancing the overall efficiency and responsiveness of the data transfer process.

However, a nuanced acknowledgment of the project's limitations is imperative. The computational intensity inherent in compression tasks remains a persistent challenge, albeit mitigated through the dynamic scaling capabilities of Kubernetes. Continuous vigilance and proactive measures are essential to fortify the system against potential security threats, emphasizing the need for perpetual advancements in security protocols.

Contributions to the field are substantial, encompassing a pioneering approach to cloud data transfer that marries message queuing and dynamic scaling. This amalgamation not only confronts existing challenges head-on but also establishes a blueprint for creating systems that are not only scalable and reliable but also secure in cloud environments.

In concluding, while celebrating the successes and strengths of our methodology, it is paramount to recognize that technological landscapes are dynamic. Future iterations of this project will inevitably involve refinements and enhancements. The collaborative nature of the cloud computing domain encourages ongoing exploration and innovation, urging us to stay agile and receptive to emerging trends and technologies.

The conclusion, therefore, marks not the termination but a transition point—a juncture from which we propel ourselves into a future where cloud data transfer systems are not only robust and secure but also adaptive and innovative in the face of evolving technological landscapes.

6.2 FUTURE SCOPE

Looking ahead, the future scope of our secure cloud data transfer project extends beyond immediate optimizations. One avenue for exploration involves the integration of edge computing to decentralize data processing, potentially reducing latency and enhancing the overall user experience. By distributing compression tasks closer to the data source, we could further streamline the transfer process.

Moreover, a future direction could involve the implementation of a user-friendly interface and dashboard, providing clients with transparent insights into the status of their compression requests, resource utilization, and overall system performance. This not only enhances user experience but also empowers users with more control and visibility into the data transfer process.

In the realm of security, ongoing efforts will be dedicated to staying abreast of emerging threats and implementing advanced encryption techniques. Continuous monitoring and regular updates to security protocols will be crucial in maintaining the robustness of the system against evolving cyber threats.

Collaborative efforts with industry partners and stakeholders can provide valuable insights and potentially lead to standardization in cloud data transfer practices. Sharing our experiences and lessons learned could contribute to a collective knowledge pool, fostering advancements in the broader field of cloud computing.

Finally, the adaptability of our methodology opens doors to integration with emerging technologies such as blockchain. Exploring the synergy between secure data transfer and blockchain's decentralized and tamper-resistant nature could pave the way for enhanced data integrity and security.

In essence, the future scope of our secure cloud data transfer project encompasses a holistic approach to innovation, spanning technological advancements, user-centric features, enhanced security measures, and collaborative efforts to shape the evolving landscape of cloud-based data processing.

REFERENCES

1. C. V. Raghavendran, G. Satish, S. V. Penumathsa, and J. M. Gummadi, "A Study on Cloud Computing Services," Int. J. Eng. Tech. Res., vol. 4, p. 67, 2016.
2. M. Attaran and J. Woods, "Cloud computing technology: improving small business performance using the Internet," J. Small Bus. Entrepren., vol. 13, pp. 94-106, 2018, doi: 10.1080/08276331.2018.1466850.
3. A. Poniszewska-Marańda and E. Czechowska, "Kubernetes Cluster for Automating Software Production Environment," Sensors, vol. 21, no. 5, p. 1910, 2021, doi: 10.3390/s21051910.
4. R. Satrio Hadikusuma, Lukas, and K. Bachri, "Survey Paper: Optimization and Monitoring of Kubernetes Cluster using Various Approaches," Sinkron, vol. 8, pp. 1357-1365, 2023, doi: 10.33395/sinkron.v8i3.12424.
5. K. Senjab, S. Abbas, N. Ahmed et al., "A survey of Kubernetes scheduling algorithms," J Cloud Comp, vol. 12, p. 87, 2023, doi: 10.1186/s13677-023-00471-1.
6. S. Vinoski, "Advanced Message Queuing Protocol," IEEE Internet Comput., vol. 10, no. 6, pp. 87-89, Nov. 2006, doi: 10.1109/MIC.2006.116.
7. F. Iqbal, M. Gohar, H. Alquhayz, S.-J. Koh, and J.-G. Choi, "Performance evaluation of AMQP over QUIC in the internet-of-thing networks," J. King Saud Univ. Comput. Inf. Sci., vol. 35, no. 4, pp. 1-9, 2023, doi: 10.1016/j.jksuci.2023.02.018.
8. S. Dixit and M. M. P, "Distributing Messages Using Rabbitmq with Advanced Message Exchanges," Int. J. Res. Stud. Comput. Sci. Eng., vol. 6, no. 2, pp. 24-28, 2019, doi: 10.20431/2349-4859.0602004.
9. P. Boisrond, "A Position Paper on Amazon Web Services (AWS) Simple Storage Service (S3) Buckets," May 2021. [Online]. Available: doi.org/10.13140/RG.2.2.17727.84640
10. M.-N. Garcia, S. Argyropoulos, N. Staelens, M. Naccari, M. Rios-Quintero, and A. Raake, "Video Streaming," in Quality of Experience, S. Möller and A. Raake, Eds. Cham: Springer, 2014, pp. 389–430. [Online]. Available: https://doi.org/10.1007/978-3-319-02681-7_19
11. A. B. Kamath and C. B. H, "A Comparison between RabbitMQ and RESTful API for Communication between Micro-Services Web Application," Int. Res. J. Eng. Technol., no. May, pp. 4665–4667, 2020, [Online]. Available: www.irjet.net.

12. C. -C. Chang, S. -R. Yang, E. -H. Yeh, P. Lin and J. -Y. Jeng, "A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning," GLOBECOM 2017 - 2017 IEEE Global Communications Conference, Singapore, 2017, pp. 1-6, doi: 10.1109/GLOCOM.2017.8254046.
13. Eng, K., Muhammed, A., Abdullah, A. et al. An Estimation-Based Dynamic Load Balancing Algorithm for Efficient Load Distribution and Balancing in Heterogeneous Grid Computing Environment. J Grid Computing 21, 7 (2023). https://doi.org/10.1007/s10723-022-09628-9
14. Lu, X., Pan, Z. & Xian, H. An efficient and secure data sharing scheme for mobile devices in cloud computing. J Cloud Comp 9, 60 (2020)
15. Peinl, Rene & Holzschuher, Florian & Pfitzer, Florian. (2016). Docker Cluster Management for the Cloud - Survey Results and Own Solution. Journal of Grid Computing. 14. 10.1007/s10723-016-9366-y.
16. Giannakopoulos, I., Konstantinou, I., Tsoumakos, D. et al. Cloud application deployment with transient failure recovery. J Cloud Comp 7, 11 (2018). https://doi.org/10.1186/s13677-018-0112-9
17. Carrión, C. Kubernetes as a Standard Container Orchestrator - A Bibliometric Analysis. J Grid Computing 20, 42 (2022). https://doi.org/10.1007/s10723-022-09629-8
18. Dang-Quang, Nhat-Minh, and Myungsik Yoo. 2021. "Deep Learning-Based Autoscaling Using Bidirectional Long Short-Term Memory for Kubernetes" Applied Sciences 11, no. 9: 3835. https://doi.org/10.3390/app11093835
19. R. Deokar, “File Transfer on Cloud using Diffie-Hellman Key Exchange in Conjunction with AES Encryption,” National College of Ireland Project Submission Sheet School of Computing, https://norma.ncirl.ie/6467/1/ravibabajideokar.pdf Dec. 2022
20. Begum MB, Deepa N, Uddin M, Kaluri R, Abdelhaq M, Alsaqour R. An efficient and secure compression technique for data protection using burrows-wheeler transform algorithm. Heliyon. 2023 Jun 23;9(6):e17602. doi: 10.1016/j.heliyon.2023.e17602. PMID: 37457815; PMCID: PMC10347677.
21. J. J. Jeya, S. Raja Ratna, G. G. Devi, M. Priya, C. Sivasankar, "An Efficient Algorithm for Secure Key Management in Cloud Environment," in Proceedings of the Eighth International Conference on Science Technology Engineering and

Mathematics (ICONSTEM), 2023, pp. 1-3, doi: 10.1109/ICONSTEM56934.2023.10142544.

22. P. SHANMUGAPRIYA, K.S.V.SRINIVAS, and RAMPAM PAVAN KUMAR, “SECURE DATA TRANSFER BASED ON CLOUD COMPUTING,” International Research Journal of Engineering and Technology (IRJET), vol. 09, no. 03, pp. 342–346, Mar. 2022.

23. Abdalla, M., Eisenhofer, T., Kiltz, E., Kunzweiler, S., Riepel, D. (2022). Password-Authenticated Key Exchange from Group Actions. In: Dodis, Y., Shrimpton, T. (eds) Advances in Cryptology – CRYPTO 2022. CRYPTO 2022. Lecture Notes in Computer Science, vol 13508. Springer, Cham. https://doi.org/10.1007/978-3-031-15979-4_24

24. Pu, Y., & Xu, M. (2009). Load Testing for Web Applications. 2009 First International Conference on Information Science and Engineering. doi:10.1109/icise.2009.720M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.

25. Draheim, D., Grundy, J., Hosking, J., Lutteroth, C., & Weber, G. (2006). Realistic load testing of Web applications. Conference on Software Maintenance and Reengineering (CSMR’06). doi:10.1109/csmr.2006.43

Cloud based Secure Data Transfer using Exchange of Cryptographic Keys

ORIGINALITY REPORT

11%	10%	5%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	ir.juit.ac.in:8080 Internet Source	3%
2	www.ir.juit.ac.in:8080 Internet Source	1%
3	journalofcloudcomputing.springeropen.com Internet Source	1%
4	medium.com Internet Source	1%
5	www.researchgate.net Internet Source	1%
6	www.arcjournals.org Internet Source	<1%
7	www.juit.ac.in Internet Source	<1%
8	dokumen.pub Internet Source	<1%
9	"Proceedings of the International Conference on Cognitive and Intelligent Computing",	<1%

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

PLAGIARISM VERIFICATION REPORT

Date: ……………………….

Type of Document (Tick): PhD Thesis | M.Tech Dissertation/ Report | B.Tech Project Report | Paper

Name: ___________________________ __Department: _________________ Enrolment No _________

Contact No. ______________________________E-mail. ______________________________________

Name of the Supervisor: ____

Title of the Thesis/Dissertation/Project Report/Paper (In Capital letters): ________________________

____

____

UNDERTAKING

I undertake that I am aware of the plagiarism related norms/ regulations, if I found guilty of any plagiarism and copyright violations in the above thesis/report even after award of degree, the University reserves the rights to withdraw/revoke my degree/report. Kindly allow me to avail Plagiarism verification report for the document mentioned above.

Complete Thesis/Report Pages Detail:

– Total No. of Pages =
– Total No. of Preliminary pages =
– Total No. of pages accommodate bibliography/references =

(Signature of Student)

FOR DEPARTMENT USE

We have checked the thesis/report as per norms and found **Similarity Index** at ……………….(%). Therefore, we are forwarding the complete thesis/report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

(Signature of Guide/Supervisor) **Signature of HOD**

FOR LRC USE

The above document was scanned for plagiarism check. The outcome of the same is reported below:

Copy Received on	**Excluded**	**Similarity Index (%)**	**Generated Plagiarism Report Details** (Title, Abstract & Chapters)	
	• All Preliminary Pages • Bibliography/Images/Quotes • 14 Words String		Word Counts	
Report Generated on			Character Counts	
		Submission ID	Total Pages Scanned	
			File Size	

Checked by
Name & Signature **Librarian**

………

Please send your complete thesis/report in (PDF) with Title Page, Abstract and Chapters in (Word File) through the supervisor at plagcheck.juit@gmail.com