Handling multilevel elasticity for distributed stream processing in cloud environment

Thesis

Submitted in fulfillment of the requirements

for the degree of

Doctor of Philosophy in Computer Science and Engineering

Submitted By

Thakkar Riddhiben Sanjaykumar

18FTPHDE29

Guided By

Dr Madhuri Bhavsar

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD-382481

June 2023

Specimen 'B': Approval Sheet

Nirma University
Institute of Technology

Certificate

This is to certify that the thesis entitled **Handling Multilevel Elasticity for distributed stream processing in Cloud Environment** has been prepared by **Thakkar Riddhiben Sanjaykumar [18FTPHDE29]** under my supervision and guidance. The thesis is her own original work completed after careful research and investigation. The work of the thesis is of the standard expected of a candidate for Ph.D. Programme in **Computer Science and Engineering** and I recommend that it be sent for evaluation.

Date: 20/6/23

Signature of the Guide

Forwarded Through:

(i) Name and Signature of the
Head of the Department (if any)

(ii) Dr R N Patel
Name and Signature of the Dean - Faculty of

20-6-23 Prof. Dr. P. N. Tekwani

(iii) Name and Signature of the Dean - Faculty of Doctoral Studies and Research

(iv) Name and Signature of the Executive Registrar
Nirma University 20/6/23

Specimen 'C': Declaration

Nirma University
Institute of Technology

Declaration

I, **Thakkar Riddhiben Sanjaykumar** registered as Research Scholar, bearing Registration No. **[18FTPHDE29]** for Doctoral Programme under the Faculty of **Dr Madhuri Bhavsar** of Nirma University do hereby declare that I have completed the course work, pre-synopsis seminar and my research work as prescribed under R. Ph.D. 3.5.

I do hereby declare that the thesis submitted is original and is the outcome of the independent investigations / research carried out by me and contains no plagiarism. The research is leading to the discovery of new facts / techniques / correlation of scientific facts already known. (Please tick whichever is applicable). This work has not been submitted to any other University or Body in quest of a degree, diploma or any other kind of academic award.

I do hereby further declare that the text, diagrams or any other material taken from other sources (including but not limited to books, journals and web) have been acknowledged, referred and cited to the best of my knowledge and understanding.

Date: 20/6/2023

Signature of the student

I agree with the above declaration made by the student.

Date: 20/6/23

Signature of the guide

Enclosure - V

Specimen 'D'

Nirma University
Institute of Technology
Certificate

This is to certify that the thesis entitled **Handling Multilevel Elasticity for Distributed Stream Processing in Cloud Environment** has been prepared by me, under the supervision and guidance of **Prof. Madhuri Bhavsar**. The thesis is my own original work completed after careful research and investigation. The work of the thesis is of the standard expected of a candidate for Ph.D. Programme. The final hard bound copy of the thesis is submitted after incorporating all the suggestions / corrections suggested by the referees.

Date: 08/02/2024

Thakkar Riddhiben Sanjaykumar

Date: 08/02/2024

8.2.24

Prof. Madhuri Bhavsar (Guide)

Forwarded Through:

9/2/24

(i) **Dr. R. N. Patel**
Dean - Faculty of Technology & Engineering

9-2-24

(ii) **Prof. (Dr.) P. N. Tekwani**
Dean - Faculty of Doctoral Studies and Research

(iii) 9/2/24
Shri G. Ramachandran Nair
Executive Registrar, Nirma University

I dedicate this thesis to my parents, brother, family, and friends for their endless love, support, and encouragement.

Acknowledgement

I am grateful to Nirma University for providing me with the opportunity to pursue my PhD and equipping me with the resources required for the research work.

I take this opportunity to express my deepest gratitude to my guide, Dr Madhuri Bhavsar (Professor and Head, CSE Department, Institute of Technology, Nirma University), for her intuitive guidance, support, and motivation. The appreciation and continual support she has imparted have motivated me to reach higher goals. Her guidance has triggered and nourished my intellectual maturity, which I will benefit from for a long time to come. She brought out the best of my abilities in the work I performed.

I would also like to express my heartfelt thanks to my research committee members, Dr. Umesh Bellur and Dr. Madhukar Potdar, for providing me with their valuable suggestions and comments during my research work.

I take this opportunity to express my sincere thanks to Dr. Rajesh N. Patel (Director, Institute of Technology, Nirma University), Dr. P.N. Tekwani (Dean, Faculty of Doctoral Studies and Research), and the PhD Section, Nirma University, for their continuous support during my dissertation journey.

I would also like to thank all faculty and staff members of the Computer Science & Engineering Department, Nirma University, Ahmedabad, for their special attention towards this work. A special thank you is expressed wholeheartedly to Dr. Vijay Ukani (Associate Professor, CSE Department, Institute of Technology, Nirma University) for his help in solving the technical error while preparing this thesis.

I am indebted and thankful to my parents, Sanjaybhai Thakkar and Alkaben Thakkar, and my brother, Dr. Parth Thakkar, for encouraging and loving me in every phase of my journey. I want to thank them for always being my greatest strength and support system and for their endless care and patience.

As one cannot mention the names of all family members, well-wishers, friends, and beloved ones, I would like to pay my regards to one and all who supported me directly or indirectly during this journey of knowledge.

Lastly, I would like to express my extreme gratitude to Almighty God for enriching me and giving me the faith and courage to pursue my dreams. The journey of this PhD has been truly, a life-changing experience. I owe my ability to pursue a PhD to the spiritual guidance and support of my family and friends.

- Thakkar Riddhiben Sanjaykumar

18FTPHDE29

Abbreviations

CSP	Cloud Service Provider
NIST	National Institute of Standards and Technology
AWS	Amazon Web Services
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
VM	Virtual Machine
SLA	Service Level Agreement
QoS	Quality of Service
DSP	Distributed Stream Processing
AWFDVP	Adaptive Worst Fit Decreasing Virtual Machine Placement
MeitY	Ministry of Electronics and Information Technology
MDP	Markov Decision Process
RL	Reinforcement Learning
EDRP	Elastic DSP Replication and Placement
ILP	Integer Learning Programming

DAG	Directed Acyclic Graph
EKF	Extended Kalman Filter
MBA	Model-Based Allocation
SAM	Slot-Aware Mapping
LSA	Linear Scaling Allocation
RSM	R-Storm Mapping
C-RANs	Cloud Radio Access Networks
RRH	Remote Radio Head
BBU	Base Band Units
ANN	Artificial Neural Network
LR	Linear Regression
LSTM	Long short-term memory
PSO	Particle Swarm Optimization
GA	Genetic Algorithm
FLNN	Functional Link Neural Network
NN	Neural Network
Bi-LSTM	Bidirectional-LSTM
GraphDF	Graph Deep Factors

AR	Auto Regression
VAR	Vector Auto Regression
SES	Simple Exponential Smoothing
RNN	Recurrent Neural Network
GRU	Gated Recurrent Unit
MA	Moving Average
ARMA	Auto Regressive Moving Average
ARIMA	Auto Regressive Integrated Moving Average
HWES	Holt-Winters exponential smoothing
MAE	Mean Absolute Error
MSE	Mean Squared Error
RMSE	Root Mean Squared Error
MAPE	Mean Absolute Percentage Error
MVMS	Multi-Variate Multi-Step resource prediction model
ML-MVMS	Multi-Level MVMS
HPC	High Performance Computing
RoI	Return on Investment
RAS	Resource Aware Strategy

RR Round Robin

List of Figures

List of Tables

List of Algorithms

Abstract

An increasing number of individuals and organizations are taking advantage of services available over the Internet due to its ease of access and constant availability. Cloud computing is a paradigm for delivering computing resources over the Internet in a highly scalable and on-demand manner. Cloud computing offers multifarious essential services to its users, ranging from infrastructure and system development environments to software as a service over the Internet. Various users consuming the cloud services to deploy different applications have their service requirements defined in a Service Level Agreement (SLA). Such applications can be real-time services, i.e., satellite data processing, banking transactions, healthcare applications, social media, etc. A cloud service provider (CSP) should deliver all its services swiftly to these applications, which demand fluctuating computational processing, on time. Real-time stream computations are perennial, receiving processing requests unpredictably and requiring a fair amount of resources for their processing in a constrained timeframe. Such a dynamic nature of applications leads to resource elasticity at runtime. In a cloud resource hierarchy, multiple resources with different processing capabilities and costs exist. In order to optimally utilize the cloud resources and ensure their uninterrupted availability for real-time processing requirements, it is required to scale the resources at each processing level efficiently. This work proposes MeghMesa, the multilevel elasticity framework in a cloud environment for processing real-time streaming applications and collectively optimizing the elasticity concern of multilevel resources while attaining SLAs and quality of service (QoS) parameters.

The MeghMesa framework consists of a multilevel, multivariable-multistep (ML-MVMS) resource forecasting and scaling module as primary functional modules. The ML-MVMS model plays a significant role in accurately identifying resources required at multiple processing levels (server, node, and operator levels) in the cloud envi-

ronment. The scaling module makes the quick allocation of resources to the volatile demand of processing, based on the outcome of the ML-MVMS model. By evaluating the proposed approach on resource utilization data of real-time streaming applications executing in a multilevel cloud environment, it is derived that the MeghMesa outperformed the existing approaches by optimally utilizing resources and quickly availing resources on demand.

Contents

Chapter 1

Introduction

With recent technological advancements, large workforces to individuals have adopted cloud computing. It has revolutionized the traditional method of computing by delivering infinite computing power without having physical resources on-premises. It offers access to an infinite pool of computing resources, networks, and storage over the Internet, managed by third-party providers. Cloud users are no longer required to invest in expensive infrastructure or bother about its management; they can leverage the on-demand computing capabilities offered by a service provider to handle dynamic workloads and process critical tasks on time. Users are charged as per the utility-based pricing model, which requires them to pay only for the services they consume. Cloud computing allows users to access the resources dynamically to process the workload efficiently without overprovisioning or underprovisioning them. Dynamic scaling of resources in real-time provides seamless functioning during increasing demand. It promises timely access to various services with high availability, security, and flexibility. Cloud computing enables quick experimentation and invention by facilitating easy access to advanced technologies, development platforms, and services.

Figure 1.1 presents the conceptual model of cloud by NIST (National Institute of Standards and Technology) (Liu et al. 2011). NIST is a non-regulatory agency

that develops and promotes standards and guidelines to foster innovations and the adoption of emerging technologies by ensuring reliability, security, and interoperability. The NIST cloud model provides a reference for CSPs to validate their services and align them as per the standards, and it also guides cloud users on how to use the cloud services. The NIST cloud model in Figure 1.1 shows deployment models distinguished based on ownership, access control, security, and sharing of the cloud resources. The deployment models are public, private, community, and hybrid.

- A public cloud is owned by a third-party service provider that delivers various services to the general public or organizations over the Internet. The public cloud service manager is solely responsible for managing the resources and availing the services. Public cloud services are remarkably elastic and economical, as users have to pay only for the resources or services they consume. Microsoft Azure, Amazon Web Services (AWS), and Google Cloud Platform are some of the public cloud service providers. They own the cloud infrastructure at distributed geographical locations, make it available to consumers over the Internet, and charge them based on a utility-based pricing model.

- A private cloud is mostly dedicated to a single organization. The infrastructure can be located on an organization's premises or managed by a third-party service provider. Access to private cloud resources is restricted to the organization only. The private cloud provides high data and computation security and more customization options for services compared to the public cloud. It is suitable for organizations requiring specialized computation needs and high confidentiality, which include government bodies, science and research institutions, and financial organizations.

- A community cloud is a shared infrastructure accessed by multiple organizations from a single community with mutual interests, such as compliance regulations,

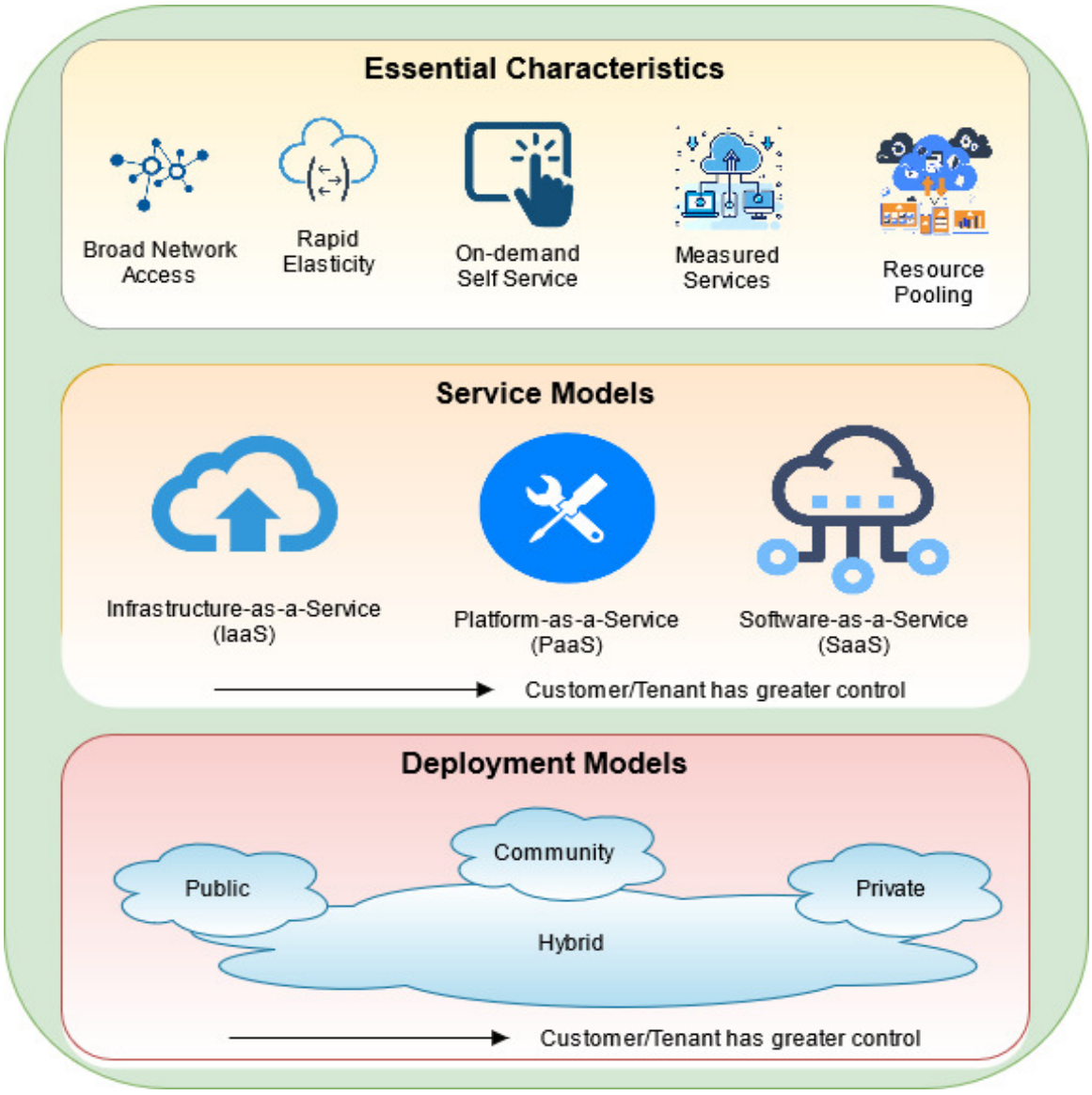


Figure 1.1: NIST cloud model (Liu et al. 2011)

security requirements, and industry standards. A community cloud is designed to achieve mission-critical objectives. The infrastructure is owned by multiple organizations within the same community or by a third party.

- A hybrid cloud combines the functionality of one or more public and private clouds; they are separate entities, however, bounded by the standard properties enabling data and application adaptability (Liu et al. 2011). The organizations can keep their data and logic in a private cloud while leveraging the public cloud's functionalities for specific processing needs. This deployment model is helpful for applications accepting dynamic workloads and requesting specific computational needs for certain applications.

The cloud computing service model refers to the various services CSP provides to users. As per (Liu et al. 2011), there are three service models in cloud computing: infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS).

- IaaS provides computing resources such as virtual machines (VM), storage, and networks to users for setting up scalable solutions on a utility-based pricing model. IaaS reduces the upfront investment in infrastructure required by businesses and individuals and also optimizes their IT spending. Virtualization is the technique for providing infinite resources that are used to handle the processing requests of users over the Internet. Virtualization provides an abstraction of computing resources from end users and applications, making use of them. In IaaS, users control the operating system, software, and data hosted on the underlying infrastructure, while CSP manages the underlying infrastructure. By leveraging IaaS services, users can reduce upfront infrastructure and maintenance costs and have more control over scalable remote resources.

- PaaS provides a platform for establishing an environment for developing, testing, and deploying any service or application. It abstracts the underlying complex infrastructure from the cloud users, including the operating system, middleware, and database. It offers a pre-configured computing environment with development tools, a runtime environment, and database systems to accelerate the production of products.

- SaaS provides ready-to-use software and services. Users can directly consume such cloud services on a subscription basis over the Internet without worrying about installation compatibility issues, maintenance, or management. The CSP is responsible for managing the internal infrastructure and platform hosting the software.

The NIST cloud model identified fundamental characteristics that each cloud service provider should deliver. They included resource pooling, on-demand self-service, rapid elasticity, broad network access, and measured service as essential properties (Liu et al. 2011).

- Resource pooling: It refers to aggregating and sharing computing resources to serve multiple users and applications. It presents an infinite pool of resources to the users. Resources include storage, memory, network, and processing capabilities. It is a fundamental principle to enable resource availability, scaling, and sharing, by isolating the execution of parallel and concurrent processes and guaranteeing high performance.

- On-demand self-service: One of the essential services which ensure the resources are provided to users without any human interactions with CSP. Users can control the access and usage of specific resources. This characteristic facilitates the user with highly available services in a smaller timeframe. It ensures the users are charged on what they-used basis, reducing their capital expense.

- Rapid Elasticity: It enables the cloud infrastructure to scale the computing resources rapidly and automatically while guaranteeing optimal throughput, cost-effectiveness and agility to the fluctuating resource demand. It enables organizations to meet their varying computing demand by seamlessly adapting the resource capacity.

- Broad network access: It refers to the cloud's ability to provide ubiquitous access to cloud services and resources from anywhere over the Internet. It also ensures to avail of services over any network which follows the standard network protocols for connecting to the cloud resources, which are geographically distributed.

- Measured service: It enables the CSP and users to monitor, measure and track the cloud services and sources used. By having detailed resource usage patterns, CSP can identify the scope of optimizing the resources and reduce the operating cost. Similarly, users can avoid unnecessary costs during periods of less usability.

In addition to these essential characteristics, cloud computing also offers security and data protection, parallel processing, availability and reliability of resources.

Among all the features provided by CSP, elasticity plays a vital role in processing dynamic workloads on distributed streaming applications. Streaming applications receive workloads at continuous and varying rates from geographically distributed locations. The input workload requires processing immediately upon arrival. Processing the bursty workload in a timely manner requires applications to have more computing resources, leading to scaling up the resources. Whereas during the calm period, fewer resources are required, leading to scaling resources down from the existing allocation. If the resources are statically provisioned for processing by considering the worst-case scenario, where resources are allocated by considering maximum processing requirements, they will be underutilized. If resources are allocated by considering average

processing requests, the performance of the application will be degraded. Hence, resource allocation is to be done based on applicable conditions.

The performance issues with static resource provisioning motivated the use of elasticity, which allows resources to be dynamically scaled up or down based on processing demand. As per (Herbst, Kounev, and Reussner 2013), resource elasticity represents the ability of the system to handle fluctuations in input workload by allocating and deallocating the resources automatically to ensure the availability of resources to process workloads at each point in time. Elasticity can be attained by resizing, migration, and replication (Galante and Bona 2012). Figure 1.2 presents how resizing (vertical scaling) and replication (horizontal scaling) of cloud resources takes place.

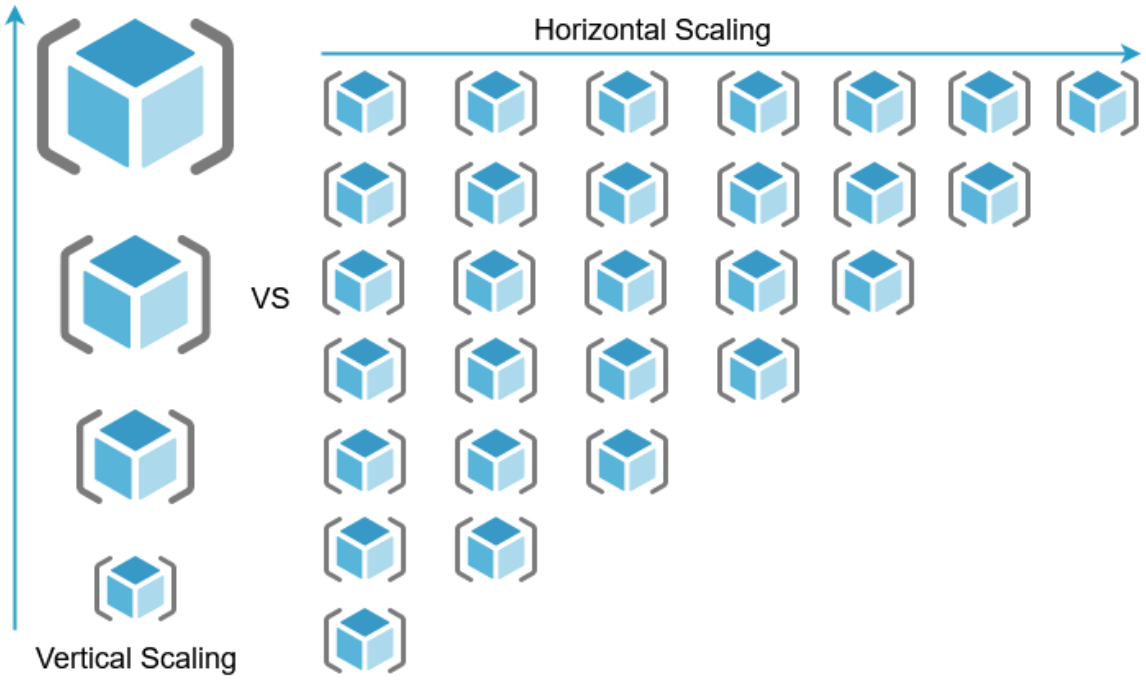


Figure 1.2: Vertical vs Horizontal elasticity (Thakkar and Bhavsar 2022)

Resizing of resources, also called vertical scaling, where the capacity or individual resources within a system are scaled up or down to handle the increase in processing

demands. In Figure 1.2, blue boxes represent the individual resources, which can be memory, CPU, or storage. The size of such resources varies depending on their type and underlying configurations. Resizing is the best solution for systems requiring strict Service Level Agreement (SLA) (Gandhi et al. 2018) and facing changes in resource demand periodically (Lorido-Botran, Miguel-Alonso, and Lozano 2014). There are a couple of shortcomings of verticle scaling:

- a resource has its capacity, and further scaling may not be cost-effective or feasible,
- many operating systems (OS) do not allow reconfiguration of resources during runtime (Rosa Righi et al. 2019)

Replication or horizontal scaling of resources solves above mentioned shortcomings by adding or removing new instances in a cloud environment. These instances may range from a physical machine, VM, or container, to any application module. Horizontal scaling performs well for systems receiving flash crowds (Lorido-Botran, Miguel-Alonso, and Lozano 2014) or sudden bursts in workload (Gandhi et al. 2018).

Elasticity can also be achieved by migrating VMs, systems, applications, or data from one physical machine or CSP to another. Migration in the cloud happens as a cause of system failure, for scaling resources, security reasons, or to adapt cost-effective solutions.

A considerable amount of work is done to optimally utilize resources while maintaining the performance of the application. However, each work differs by responding differently to Why, When, and How to scale resources.

In response to the question 'Why' resource scaling, it refers to the underlying motivation for resource scaling. Literature proposed solutions to pursue cost efficiency or enhance the application's throughput and latency. Heinze et al. 2015 proposed a reactive online parameter optimization model for elastic DSP (distributed stream

processing), which allows users to define the desired quality of service in terms of threshold values and parameters. This model optimizes the parameters to achieve a trade-off between monetary cost and latency (Mencagli 2016) proposed a mechanism that uses a game theory approach for the distribution of control logic among local modules for deciding the resource consumption of individuals. Through implementation, the authors prove that the proposed mechanism improves efficiency, along with performance, and reduces operating costs. Liu and Buyya 2017 proposed a heuristic-based resource-efficient scheduling approach to reduce inter-node communication, which improves the latency.

In response to 'When' to perform resource scaling, which refers to the target time to scale resources for handling dynamic workload conditions, the literature has described proactive and reactive approaches. Bibal Benifa and Dejey 2019 proposed a proactive resource allocation algorithm that learns the environment and performs the resource distribution accordingly, improving response time, CPU utilization and throughput. Shekhar et al. 2018 proposed a proactive, vertical scaling approach for latency-sensitive applications. Hidalgo, Wladdimiro, and Rosas 2017 proposed a solution that scales processing operators in proactive or reactive order, according to the data stream flow, to utilize processing elements in elastic stream processing.

In response to 'How', which refers to the way resource scaling is performed, the literature used a migration option to relieve overloaded and underloaded resources, and a replica management option for handling the parallelism of processing operators. Nashaat, Ashry, and Rizk 2019 proposed an algorithm that clusters VMs according to their CPU and memory parameter values and places them on physical machines using the adaptive worst fit decreasing virtual machine placement (AWFDVP) algorithm. This approach aims to reduce excessive VM migration, which introduces instability and increases data transfer, leading to performance degradation. The proposed algorithm performs smart VM migration, reducing the number of VM migrations and

power consumption and improving overall performance. Noshy, Ibrahim, and Ali 2018 discuss various live VM migration techniques that target memory management issues. Lombardi et al. 2017 proposed an algorithm that proactively and reactively manages the parallelism of a number of operators to optimize available resources.

To answer the Why, When, and How to scale resources, the literature elaborated on elasticity in the context of applications and abstracted issues related to processing platforms. Existing literature homogeneously focuses on execution resources and misses the hierarchy of execution resource containers (operator, VM, physical infrastructure) onto which application modules are mapped. A container is any entity that provides processing elements to a computation (Marangozova-Martin, De Palma, and El Rheddane 2019).

Cardellini et al. 2018 designed the elastic DSP replication and placement (EDRP) framework for QoS-aware resource placement and replication in a geo-distributed environment. Sun et al. 2020 proposed the Dr-Stream, the dynamic framework for redirecting real-time streaming data to computing resources to advance the latency and throughput in the stream processing system. Russo Russo et al. 2018 proposed a multi-level adaptation solution at the application and infrastructure levels, by significantly reducing resource wastage and negligible application performance degradation. However, they considered infrastructure and application level scaling, separately. Herbst, Kounev, and Reussner 2013 stated that while adapting the resources to fulfil the processing requirement, multiple types of resources are required to be scaled up or down, where each of the resources has different elasticity properties. They also stated that elasticity could be considered at multiple levels if the resource type is a container of other resources.

Many studies on elasticity fail to consider the impact of resource scaling at different levels of processing containers concurrently. On the other hand, Marangozova-Martin, De Palma, and El Rheddane 2019 targeted the resource elasticity at multiple execu-

tion containers. However, they considered static resource elasticity and a reactive approach to processing incoming workloads, which cannot handle the fluctuations in workload generated by real-time streaming applications. Real-time applications are ones that demand volatile computational resources to immediately analyze and process large volumes of real-time data streams generated from various sources, such as sensors, devices, and applications. In a solution to efficiently handle dynamic real-time applications executing in the cloud environment and optimally utilizing the available resources, this work targeted pro-active elastic scaling at multiple processing containers simultaneously in a cloud hierarchy. This work also ensures that the performance of real-time applications is not affected and that Quality of Service parameters are achieved as per the service level agreement (SLA).

In this research work, three different containers from the cloud framework are considered for scaling. The containers are referred to as different processing levels, each having a different amount of computing resources. The server, node, and operator are three levels, each containing processing elements inside them. The server level is the topmost level in the cloud hierarchy, containing multiple different configurations of nodes. The node level is divided among multiple operators, where actual processing takes place. The server level is the highest processing level, whereas the operator level is the lowest in a cloud hierarchy. Each server, node, and operator in the respective processing levels have different amounts of CPU and memory shares. Along with that, the server level contains the count of nodes executing inside it, and the node level contains the count of operators executing inside it.

1.1 Distributed stream processing

The ease of access to the uninterrupted availability of the Internet and the worldwide acceptance of information technology in the areas of engineering, healthcare, government, business, agriculture, and scientific study have resulted in an explosion

of data. In general, all of these domains are required to collect, process, and analyze the data streams to extract valuable information and detect patterns and outliers. Stream processing is a computing model that enables data analysis in a scalable and efficient manner. It can perform multiple operations on the incoming data streams, serially or parallelly. This functioning, starting from data generation to its processing and delivery to the store, is called the stream processing pipeline. The pub/sub (publisher/subscriber) and source/sink are the fundamental paradigms for processing streaming data. The source or publisher is a generator of data streams (i.e., satellite signals, financial transactions, application logs) processed by stream processing systems, and the results are stored in the sink or consumed by subscribers (Thakkar and Bhavsar 2022).

Distributed Stream Processing (DSP) is a system that processes an unconstrained data flow in real-time and extracts critical information, which assists in addressing the situation in real-time. The way DSP processes applications is represented by a directed acyclic graph (DAG), where each vertex presents different tasks and the edges transfer the dataflow between the tasks (Eskandari et al. 2021). The DAG contains input streams, operators as processing vertices, and consumers as storage vertices. The operators continuously process input data and produce outgoing streams, which are processed by other operators or stored by the sink operator.

Various stream processing tools and frameworks have been designed to analyze and process large volumes of real-time streaming data generated from various sources, such as remote sensing devices, scientific, wearable, mobile, and edge devices and applications (Assuncao, Silva Veith, and Buyya 2018). Some frameworks use a dataflow approach, where incoming data is processed as streams and redirected through a directed graph of operators residing on distributed hosts that apply application logic. On the other hand, some frameworks employ discretizing incoming data streams by temporarily storing them in smaller time windows and then performing micro-batch

operations on the stored data. The latter approach improves the fault tolerance and scalability of stream-processing frameworks by managing slow-processing tasks more efficiently. However, the first approach can quickly respond to the data streams upon their arrival (Assuncao, Silva Veith, and Buyya 2018). Stream processing frameworks process real-time workloads parallelly and efficiently with high fault tolerance and resilience. Apache Storm and Apache Samza are real-time processing frameworks. There are also other frameworks for processing the stored data at regular intervals, called batch processing frameworks. Apache Hadoop is one of the most widely used batch-processing frameworks. Several hybrid frameworks are available that process the data in real-time and in batches to meet the business requirements. Table 1.1 lists various stream processing frameworks with their processing categories.

Table 1.1: Stream processing frameworks

Name	**Category**
Apache Haddop (Foundation 2022b)	Batch processing
Apache Flink (Foundation 2022a)	Batch processing
Apache Storm (Foundation 2022d)	Real-time processing
Apache Samza (Foundation 2019)	Real-time processing
Apache Spark (Foundation 2021a)	Hybrid processing

In order to improve scalability, many of these frameworks have been deployed on the cloud, which provides elastic scaling of resources based on processing demands. However, as per (Assuncao, Silva Veith, and Buyya 2018), it is challenging to make the processing of streaming applications elastic. Thus, to provide a highly elastic environment for streaming applications executing on a cloud environment, MeghMesa,

the multilevel elastic framework, is designed in this work. The proposed framework also ensures the optimal utilization of available cloud resources.

1.2 Motivation

Cloud architecture consists of a hierarchy of various kinds of resources. Each of them can be seen as a separate dimension of the adaptation process with its elasticity properties (Herbst, Kounev, and Reussner 2013). However, each processing level includes resources such as memory, CPU, and network. It is significantly challenging and complicated to predict the exact requirements of each resource (Khan et al. 2022) for real-time application processing. Such complexity in a cloud environment motivates designing and developing a framework that dynamically scales multidimensional resources based on real-time processing demand at each processing level.

1.3 Problem definition

The research work is envisioned to design a framework for managing the multilevel elasticity of distributed stream processing (DSP) in a cloud environment. DSP systems cater to computational analysis and support transformation or value extraction operations on data deluge received from distributed data sources, i.e., satellites, mobile devices, large-scale scientific environments, etc. To keep up with the high volume and velocity of data, applications executing on DSP systems need to elastically scale their processing on different levels of computing resources (operator, node, and server) to achieve QoS parameters and fulfil the SLA. Thus, it is required to accurately predict resources which can lead to efficient scaling decisions required at multiple processing levels in a cloud environment.

1.4 Objectives

- To design an architecture facilitating elasticity for supporting distributed stream processing applications.
- Design and develop multilevel elastic model functioning at operator, node and infrastructure level.
- Achieve optimized resource utilization during elastic-scaling of cloud resources supporting distributed stream processing applications.

1.5 Research scope

The scope of this thesis is to design a multilevel elastic framework for a cloud environment for processing real-time streaming applications. The proposed framework contains a resource prediction model to get knowledge of the precise amount of resources required while processing the fluctuating workload efficiently. Based on the prediction of resources, the scaling decision is taken at each processing level.

1.6 Contributions

- The multilevel elastic framework, MeghMesa, is designed for a cloud environment for processing real-time streaming application workloads.
- The multilevel elastic model, ML-MVMS (Multi-Level - Multi-Variate Multi-Step), is proposed, which efficiently identifies the resources required in advance to handle dynamic workloads.
- The performance of ML-MVMS is evaluated by comparing it with existing approaches and using evaluation metrics such as MAE (Mean Absolute Error), MSE (Mean Squared Error), and RMSE (Root Mean Squared Error). Based on

the availability of resources at the respective levels, resource scaling decisions are taken to optimally utilize them.

- MeghMesa has been evaluated on a real-time experimental setup, and its performance is validated by comparing it with existing approaches.

1.7 Summary

This chapter discusses the basic concepts of cloud computing, including various characteristics, deployment models, and services. The elasticity property of a cloud is elucidated for forming the base of this research work. As this work targets elasticity for distributed streaming applications, stream processing was also discussed. This chapter also introduced various stream processing platforms, which process streaming data in parallel. Based on the domain analysis, a research problem was defined, and the objectives were decided to attain it.

1.8 Roadmap of the Thesis

The rest of the thesis is organized by the collection of chapters as below:

Chapter 2: Literature review

This chapter discusses the existing work for attaining elasticity at different execution levels in a cloud environment while processing the real-time streaming workload. The domain is analyzed in three parts (threefold objectives). First, the distributed stream processing framework in a cloud environment and approaches for attaining elasticity are discussed. The second part explores the methods for dynamically scaling the resources at multiple levels for the real-time streaming workload in cloud environment. Then the elasticity techniques used for resource forecasting are discussed.

Chapter 3: Methodologies and dataset for proposed framework

This chapter presents the methodologies for implementing elasticity at multiple processing levels of cloud architecture while efficiently operating distributed streaming applications. First, the distributed stream processing platforms, which process real-time streaming applications, are elucidated. Then, the different approaches for accurately forecasting the resource consumption for attaining elasticity at the server, node, and operator levels are discussed.

Chapter 4: Design and development of MeghMesa framework

This chapter includes a detailed discussion of the development of the proposed framework. The conceptual diagram demonstrates the birds-eye view of the proposed framework, which is discussed in depth in an architectural view. The subsequent part of the chapter reviews the individual modules of the architecture. Then, the forecasting model with hyperparameter selection for accurately estimating the resource requirement in real-time is discussed.

Chapter 5: Implementation, evaluation and performance analysis

This chapter discusses the results obtained by executing the proposed framework in a real-time environment. Then the performance of the ML-MVMS forecasting model and the MeghMesa framework was examined by comparing them with existing works.

Chapter 6: Conclusion and Future scope

This chapter concludes the thesis with insights and findings, along with future research directions.

1.9 Indian Studies

In recent years, cloud computing has gained significant momentum in India, driven by the country's rapid digital transformation, increasing Internet penetration, and the need for scalable and cost-effective IT solutions. The Government of India has recognized profound opportunities in cloud computing and has implemented several policies and launched some initiatives to transform India into a digitally empowered country. The Government of India launched Meghraj - the Cloud Computing initiative for delivering e-services in the country for faster development and deployment of eGov applications, while efficiently utilizing ICT spending. The Ministry of Electronics and Information Technology (MeitY) has created a reference architecture for guiding various departments in government to build their cloud deployment architecture with recommended components and activities. The National Digital Communications Policy envisioned in 2018 to set the goals, initiatives, strategies and intended policy outcomes for achieving digital empowerment and advancing the living standards of Indian citizens. The Indian government has initiated many cloud-based projects, including Aadhaar, the Aarogya Setu app, and DigiLocker.

Many Indian researchers contributed to the field of cloud computing to improve the quality of various cloud services delivered to users as well as the profitability of service providers.

Shukla and Simmhan 2018 proposed a model-driven schedular for task-to-resource mapping to improve the performance of DSP applications. The authors experimentally stated that the proposed approach reduces resource consumption as compared to the existing approach for the same input workload.

Gandhi et al. 2018 presented a solution to optimally utilizing the resources scaling by modelling the workload characteristics and quantitatively deciding the scaling options. The authors stated that this work effectively reduces the resources required

to process the input workload, while attaining the SLA.

Thakkar and Bhavsar 2022 identified that the elasticity in a cloud environment is attained at individual processing levels. However, the authors proposed a novel approach to concurrently manage elasticity at multiple processing levels. They claimed that this approach would improve the utilization of computing resources efficiently without degrading the performance of applications.

Bhatia et al. 2019 presented complete work on the SDN-based VANET (SDVN) system as a whole, with its architecture, use cases, scope of opportunities, and challenges.

Chapter 2

Literature review

This chapter discusses the existing work carried out for attaining elasticity at different execution levels in a cloud environment while processing the real-time streaming workload. The domain is analyzed in three parts. First, the distributed stream processing framework in a cloud environment and approaches for attaining elasticity are discussed. The second part explores the methods for dynamically scaling the resources at multiple levels for the real-time streaming workload in a cloud environment. Then the elasticity techniques used for resource forecasting are discussed.

2.1 Distributed stream processsing in cloud computing

Distributed stream processing (DSP) frameworks process real-time workloads efficiently and parallelly upon arrival. As the number of operators to process incoming workload changes frequently, it requires enough computing resources, which demands elasticity of resources. Thus, to get the high availability and elastic properties of resources, DSP systems are placed on the cloud.

Sun et al. 2020 stated that it is challenging to decide a way to continuously adapt

resource adjustments for processing the fluctuating data streams. As the rescheduling approach does not work due to the longer decision time, frequent changes to the underlying computing environment or the loss of vertex state information may occur. To address these problems, the authors proposed the Dr-Sream framework, which processes dynamic workloads in a scalable and elastic manner while improving system performance and mapping input workloads to resources without causing data loss. Dr-Stream contains four stages: 1) Topology construction: based on the user logic, the topology structure is designed and submitted to the stream processing environment; 2) Instantiation: number of instances determined for each vertex to balance the incoming workload; 3) Scheduling: This stage determines the scheduling of topology onto the available computing nodes, while optimally utilizing resources. The modified first-fit approach is applied to the current deployment of vertices. 4) Redirection and Rescheduling: A lightweight load balancing approach to balance the data center load is proposed, in which factors influencing the load state of vertex instances are reduced to only n instances of that vertex. A logical ring-based approach is proposed for storing states of stateful verities. Results show that the Dr-Steam framework performs better than the default strategy of apache storm by providing high throughput, low latency, a lower average load balancing value, and a low average load ratio.

Russo, Cardellini, and Presti 2019 stated that current work discusses the dynamic scaling of homogeneous computing resources. However, the data center may contain various resources that cloud users demand to process different applications. The authors proposed a Markov Decision Process (MDP) based solution for controlling elasticity on heterogeneous resources. MDP-based solution suffers from limited space issues. Also, it requires complete knowledge of the system to provide a timely response in terms of computing resources for incoming data streams. They addressed these issues by blending reinforcement learning (RL) and function approximation (FA)

techniques. Since RL is a self-learning technique, it takes longer to converge when the size of the system increases. By providing an approximate state space representation, FA techniques achieve near-optimal solutions by reducing memory and computing resources. With the blending of model-based (MDP) and model-free (RL) methods, the authors presented that this integration improves the convergence problem of MDP with limited system dynamics. From the experiments, the authors mentioned that with a priori knowledge of the system, the proposed solutions could effectively work for improving the dynamic scaling online.

Cardellini et al. 2018 optimized resource heterogeneity while proposing an elastic DSP replication and placement (EDRP) framework for providing QoS-aware replication and placement of applications and resources in a geo-distributed environment. The EDRP framework is formulated as an integer learning programming (ILP) problem to optimize deployment at run-time while considering monetary and reconfiguration costs and response time. It is used to minimize the response time in terms of choosing the critical path for reducing the traversal of the application DAG and the monetary cost of computing and networking resources used for operating the input data streams of the application. From the evaluation, the authors proved that system downtime could be reduced by ten times by considering resource reconfiguration costs.

Gedik et al. 2013 addressed profitability challenges with the auto-parallelization of conventional distributed stream processing systems. The authors attain the elasticity of DSP applications by dynamically parallelizing the regions of the application's directed acyclic graph (DAG) for handling fluctuating data streams. Replication of DAG regions allows the processing of big data in parallel.The control algorithm is executed periodically to collect the congestion index and throughput data, which are then utilized to ascertain the optimal number of parallel channels needed to achieve high throughput and maximize resource utilization. A state migration algorithm is exe-

cuted to change the number of channels in a stateful parallel region, and data splitting is done with a round-robin algorithm when a parallel region is stateless, while a hash-based algorithm is in either case. The proposed approach ensures the security of data and maintains data arrival orders during delivery, regardless of whether migration occurs. The experiment demonstrates that the proposed approach achieves increased throughput, a rapid settling period, and mitigates oscillation and overshooting.

Lombardi et al. 2017 proposed ELYSIM, a fine-grained model that estimates the resource requirement for DSP applications by enabling the independent scaling of two symbiotic entities: operators and resources. ELYSIUM can process in reactive and proactive modes, depending on the type of input load it is operating over. When real-time input load is used, it scales reactively, and when input load is predicted over a specific prediction horizon, it scales proactively. Q-Learning is used to automate parameter tuning. By experimenting with the proposed approach on the DSP framework, the authors concluded that the ELYSIUM approach enhances resource utilization in comparison to the joint scaling approach through resource economization.

Borkowski, Hochreiner, and Schulte 2019 aimed to reduce the number of scaling operations needed to process incoming data. They proposed a concrete scaling mechanism based on an Extended Kalman Filter (EKF). The proposed approach identifies CPU and memory consumption, referred to as intrinsic measures, for incoming data, referred to as extrinsic measures. With this mapping of intrinsic and extrinsic measures, the filter can quickly identify resource demand and make optimal scaling decisions.

Garí, Monge, and Mateos 2022 proposed a q-learning-based approach to dynamically select the scaling policies by considering the workflow dependencies. The authors are targeting the auto-scaling of scientific workloads in a cloud environment by reducing the economic cost and execution time. The experimental evaluation deduces

that giving importance to the execution time gives the optimal result.

The study found the significance of elasticity in a cloud environment for real-time streaming applications. Achieving elasticity in real-time streaming applications presents several challenges, and this chapter provides an overview of existing approaches to address these issues. The key findings highlight the significance of elasticity in cloud computing and real-time streaming applications. The next section discusses the significance and challenges of attaining multilevel elasticity in a cloud environment.

2.2 Multilevel elasticity

Elasticity is the ability of a system to scale up and down resources to process a dynamic workload. Herbst, Kounev, and Reussner 2013 stated that an adaptation process involves scaling up or down multiple types of resources, each with its own elasticity properties, and if a resource type is a container of other resources, elasticity can be considered at multiple levels. In order to optimally utilize the cloud resources, it is required to scale the resources at each of these processing levels efficiently.

Marangozova-Martin, De Palma, and El Rheddane 2019 experimentally demonstrated that provisioning the low-cost processing container improves the performance; however, provisioning the high-cost processing container reduces the performance. The authors considered threads as low-cost processing containers and virtual machines as high-cost processing containers. In this work, the workload on each operator is measured at regular intervals, and a scaling operation is performed in case congestion is found. The experiments (Marangozova-Martin, De Palma, and El Rheddane 2019) demonstrated that provisioning the wrong execution container (thread, process, and virtual machine) leads to performance degradation. The proposed approach is application specific and reactive; thus, it is necessary to develop a dynamic approach that will work with any kind of application.

Russo Russo et al. 2018 proposed a hierarchical solution for autonomously controlling elastic DSP applications and infrastructures that significantly reduces resource wastage with negligible application performance degradation. The authors stated that the data and application parallelism should be elastically adapted at runtime to match the workload and prevent resource waste. They proposed the Multi-Level Elastic and Distributed DSP Framework (E2DF), which consists of a two-level hierarchical solution where centralized components coordinate subordinate distributed managers, which locally control the elastic adaptation of the application components and deployment regions. For designing a self-adaptive strategy, the authors utilized an RL-based approach. From experimental evaluations, the author stated that the E2DF optimally utilize the resources specified by the user by autonomously identifying the requirements.

Shukla and Simmhan 2018 experimentally demonstrated that there is no linear relation between the flow of input rate and CPU as well as memory requirements. Thus, to meet the performance requirements of applications, DSP systems are required to effectively schedule the data flow based on the available resources. The authors propose a model-driven scheduling approach for DSP systems. The scheduler contains two parts: 1) resource allocation: which determines the degrees of parallelism for each task and the number of computing resources required for dataflow; 2) resource mapping: which determines the mapping of threads to resources. The authors claimed that thread and resource slot allocation for DAG and task thread to resource slot mapping had yet to be attended concurrently by any existing work. The model-based allocation (MBA) algorithm is proposed for optimal resource allocation, and the slot-aware mapping (SAM) algorithm is proposed for thread-to-slot mapping. MBA identifies the peak input data flow processed by one resource slot, the count of threads for the specific task, and the CPU and memory required for the task. SAM first collects and bundles the available threads. The total number

of threads in a bundle is the maximum number of threads at the peak input rate that can be supported by the task in an empty slot. By comparing the performance of the MBA+SAM combination with the existing Linear Scaling Allocation (LSA) + R-Storm Mapping (RSM) combination, the authors inferred that MBA allocates an optimal resource slot for a fixed input rate and SAM effectively assigns a bunch of task threads to resource slots recommended by MBA; however, LSA allocates an extra resource slot for the same input rate, and RSM also demands more slots than proposed by LSA. The proposed scheduling approach only provides resource allocation for data flow before application execution, not considering real-time data flow. Thus, it fails to handle the dataflow fluctuations in real time.

The task co-location challenge identified by (Shukla and Simmhan 2018) has been pointed out in (Eskandari et al. 2018) for optimally utilizing the resources. This problem can be addressed by decreasing data transmission between communicating tasks. In work by Eskandari et al. 2018, a heuristic scheduling approach known as T3-Scheduler is introduced. This approach determines the communication of tasks among each other and locates them to a computing node for efficiently utilizing the targeted node. T3-scheduler effectively works in both cloud and heterogeneous fog environments. Through experimentation, the authors deduced that the proposed approach outperforms existing approaches.

Farrokh et al. 2022 introduced SP-ant, an innovative operator scheduler based on ant colony optimization, designed to address the challenges posed by heterogeneity in cluster environments. By leveraging the natural behaviour of ant colonies, SP-ant intelligently assigns operators to computing nodes, by efficiently handling load balancing, communication costs, and node capabilities. The results presented that the proposed approach outperformed existing approaches by reducing processing latency, enhancing performance, and improving overall system efficiency.

Russo Russo, Cardellini, and Lo Presti 2023 proposed a two-layered control architecture-

based autoscaling solution for processing DSP applications on heterogeneous infrastructure. The authors devised reinforcement learning for handling model uncertainty at the bottom layer and Bayesian optimization techniques for handling model uncertainty at the top layer. The results show that the proposed approach is able to improve performance in terms of quick response time and improved resource usage while processing dynamic workloads.

Rosa Righi et al. 2019 proposed an Elastic-RAN approach targeting multi-level elasticity for Cloud Radio Access Networks (C-RANs). C-RANs architectures are designed to overcome the issues related to system updates and administration in traditional RAN. It was proposed to leverage the flexibility of distributed systems and the elasticity of cloud computing. C-RAN consists of a remote radio head (RRH), which performs analog radio frequency functions; baseband units (BBU), for processing the digital signals; fronthaul, referring to a connection between BBU and RRH. The key idea behind C-RAN was to virtualize the functionalities of BBU. However, it is not leveraging cloud elasticity. Thus, Elastic-RAN is proposed in this work to improve performance and provide a malleable resource rearrangement for C-RANs. It provides non-blocking multi-level elasticity and adaptive elasticity grain functionalities. The multi-level elasticity refers to coordinating resources in BBU pools and BBUs. Elastic-RAN provides non-blocking services to users by allowing fine-grained resource adaption. As the proposed approach targets telecom operators, attention is paid to providing cost-efficient and customer-centric functionalities. The experiments proved that the proposed elasticity controller quickly reacts to load variations.

It is concluded that the elasticity at various processing levels has been studied separately in the literature. However, it is necessary to handle the elasticity concurrently at multiple processing levels in a cloud environment while efficiently executing the real-time streaming applications and optimally utilizing the resources.

2.3 Resource forecasting techniques in cloud environment for attaining elasticity

Real-time applications receive an unbounded sequence of data flows required to process immediately. Such applications executing in the cloud environment call for sufficient resources for their execution. The general resource provisioning strategy includes static allocation, which leads to over- and under-utilization of resources. Thus, having operational awareness of the resources to be consumed is necessary to build a holistic resource provisioning model for the upcoming computational demand. The significant difference in the resource utilization trend in the recent technological era has dramatically transformed resource prediction approaches. This section includes the various resource forecasting techniques used by other researchers to attain various objectives.

Moreno-Vozmediano et al. 2019 presented ML-based time-series forecasting and queuing theory-based approaches for dynamic scaling of computing resources. The authors predicted resource demand with the SVM regression model for the scaling decision. It is combined with the M/M/c queuing model to predict the exact number of resources based on the expected load. From the results, the authors claimed that it performed better than the simple methods.

Borkowski, Schulte, and Hochreiner 2016 proposed an artificial neural network (ANN) based holistic resource provisioning model. As the authors used an offline learning approach for model training, the model can not identify the resource requirements for the unseen data flow that may occur during real-time prediction.

Singh, Gupta, and Jyoti 2019 proposed Technocrat cloud provisioning architecture consisting of an application provisioner, performance modeler, and workload predictor. The workload predictor was designed using linear regressor (LR), ARIMA, and SVR models for forecasting non-stationary workloads for web applications. ARIMA

is used to predict the fast-scale workload, whereas LR and SVR predict the slow-scale workload.

Thonglek et al. 2019 used an RNN-based LSTM model to predict resource allocation for a given job based on historical data. Two-layered LSTM discovers the relationship between resource usage and the allocation of resources (CPU and memory). From the experiments, the authors identified that increasing the size of the memory cell enhances prediction accuracy at the cost of a longer training time.

To improve resource prediction accuracy, Zhu et al. 2019 presented the long-term, short-term memory (LSTM) encoder-decoder network with an attention mechanism. The attention mechanism signifies the parameters that greatly influence prediction results by giving them greater weight. However, the authors deduced that the attention mechanism does not substantially impact the performance of the results.

Chudasama and Bhavsar 2020 proposed a Bi-directional LSTM model and a queuing theory-based short-term approach to predict resources for one hour based on historical resource utilization.

Malik et al. 2022 stated that the existing resource utilization mechanisms consider single resource prediction while overlooking the relation among the different resources. They proposed a hybrid model named FLGAPSONN consisting of GA-PSO (genetic algorithm-particle swarm optimization) for training the model with high accuracy and FLNN (functional link neural network) for predicting multi-resource utilization. For multivariate resource prediction, Xu et al. 2022 presented the sliding window, S-MTF, and esDNN algorithms. S-MTF converts multivariate time-series data into a supervised learning time series, and esDNN predicts future resource usage with a modified GRU model. The experimental results demonstrated that this approach outperformed state-of-the-art algorithms and improved resource provisioning. Kumar and Singh 2018 presented a workload prediction model based on a self-adaptive differential evolutionary algorithm and ANN. The proposed model uses an evolutionary

algorithm to reduce the initial parameter selection effect during training. The results show that the proposed model beats the other models with a substantially lower mean squared prediction error.

Tran et al. 2018 proposed multivariate fuzzy LSTM (MF-LSTM), a proactive auto-scaling service for processing multivariate monitoring data. The fuzzification technique employed in this work addresses the need to analyze metrics correlation. In this approach, the relationships among parameters are identified at every prediction window, which is not possible when dealing with a real-time workload.

Mason et al. 2018 implemented an evolutionary neural network (NN) method for forecasting CPU utilization and reducing energy efficiency while performing adaptive resource scaling. The implementation outcome demonstrated that the Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) converges quickly to the best solutions on the training data. However, differential evolution (DE) provides statistically comparable results to CMA-ES on test data. However, the accuracy of the network decreased while predicting multiple future steps.

For predicting single-step host load, Song et al. 2018 utilized the LSTM model. The proposed model efficiently schedules resource allocations and optimally utilizes them.

Karim et al. 2021 proposed the BHyPreC architecture for predicting CPU utilization for the future. The BHyPreC consists of bidirectional-LSTM (Bi-LSTM) on top of the stacked LSTM and GRU. The authors observed the effect of different prediction and history windows on the model's performance. They used the Bitbrains Shen, Van Beek, and Iosup 2015 dataset for model training and evaluation. From the results, the authors show that the BHyPreC outperforms other state-of-the-art techniques for short- and long-term future prediction.

Rahmanian, Ghobaei-Arani, and Tofighy 2018 proposed a learning automata (LA) based, ensemble resource utilization forecasting framework. The LA theory is used to

tune the weights of the prediction model. The proposed framework provides precise resource prediction in a virtualized cloud environment.

Chen et al. 2021 proposed Graph Deep Factors (GraphDF), a graph-based deep hybrid probabilistic forecasting model. It is a relational global model capable of learning complex non-linear time-series data patterns universally with the graph structure. The key objective of GraphDF is to improve prediction accuracy and computational efficiency.

Table 2.1 summarises the existing work for resource prediction in a cloud environment.

As the demand for cloud resources fluctuates continuously, it is required by the CSP to have a pool of resources at multiple processing levels in a cloud hierarchy to be prepared before processing demands arrive. Thus, it is required to predict the resource demand in advance to cope with the volume of data flow for processing. The literature was studied to identify the different methodologies/approaches for attaining elasticity for multiple parameters and multiple timestamps in advance. Thus, the Multi-Variable Prediction and Multi-Step Prediction columns in Table 2.1 represent whether the given literature considers multiple variables and prediction steps while forecasting resource demand.

Paper	**Model Used**	**Dataset**	**Model Evaluation Metrics**	**Error metrics**	**Multi-variable Prediction**	**Multi-Step Prediction**
Borkowski, Schulte, and Hochreiner 2016	ANN	Travis CI and GitHub	Per-task duration of different tasks,	RMSD	×	×
Shyam and Manvi 2016	Bayesian Model	AmazonEC2, Google CE-DataCenters (Reiss et al. 2012)	vCPU instance	MSE	×	×
Prasad and Bhavsar 2020	Reinforcement Learning, LSTM	Bitbrains (Shen, Van Beek, and Iosup 2015)	CPU utilization, Disk read/write Throughput and Memory utilization	MAE, MSE, RMSE	✓	×

Chudasama and Bhavsar 2020	Queuing theory, Bidirectional LSTM	Private Cloud dataset	Workload of a web server	MAE, MSE, RMSE	×	✓
Singh, Gupta, and Jyoti 2019	LR, ARIMA, SVR	ClarkNet, NASA	HTTP requests	MAE, MSE, RMSE, MAPE	×	×
Thonglek et al. 2019	LSTM	Googles cluster-usage trace (Wilkes 2011a)	Requested CPU and memory resource, Used CPU and memory resource	-	✓	×
Kumar and Singh 2018	Evolutionary Algorithm	NASA, Saskatchewan	Number of HTTP requests	RMSE	×	✓
Yang et al. 2018	Novel admission control mechanism	Classic MapReduce	CPU Utilization	Makespan	×	×

Mason et al. 2018	Evolutionary Neural Networks (NN)	PlanetLab workload trace (Calheiros et al. 2011)	CPU utilization	MAE, MSE	×	✓
Tran et al. 2018	MF-LSTM	Googles cluster-usage trace (Wilkes 2011a)	CPU and Memory usage	MAE	✓	×
Song et al. 2018	LSTM	Googles cluster-usage trace (Wilkes 2011a), Traditional Distributed System Dinda 1996	CPU Usage	MSE, MSSE	×	✓
Karim et al. 2021	BHyPreC: Bi-LSTM Based Hybrid RNN	Bitbrains (Shen, Van Beek, and Iosup 2015)	CPU Usage	MAE, MSE, RMSE, MAPE	×	✓

Chen et al. 2021	GraphDF: Graph Deep Factors	Google Trace, Adobe Workload Trace, Graph Construction	CPU Usage	MAPE	×	✓
Rahmanian, Ghobaei-Arani, and Tofighy 2018	Learning automata	CoMon project (Park and Pai 2006)	CPU Usage	RMSD, error ratio, absolute Error	×	✓
Malik et al. 2022	FLGA PSONN	Google cluster workload traces (Reiss et al. 2012)	CPU and memory Usage	MAE	✓	×

Xu et al. 2022	GRU	Alibaba, Google cluster workload traces (Wilkes 2011b)	CPU and memory Usage	MSE, RMSE, MAPE	✓	×

Table 2.1: Summary of existing works for resources prediction (Thakkar, Thakkar, and Bhavsar 2023)

2.4 Challenges for prediction of resource usage

This section discusses the challenges that must be accomplished by a CSP while designing an accurate resource prediction model. Figure 2.1 highlights significant challenges for resource prediction in a dynamic cloud environment.

- Complexity: The model should be less complex but able to identify the hidden patterns in the data.
- Versatile: The model should be versatile enough to forecast future resource requirements for diverse users, encompassing resources characterized by varying configurations.
- Risk parameter: It should mitigate the effect of parameters causing the risk of over-provisioning and under-provisioning.
- Cost: The model should be less expensive in terms of time and computing.

Figure 2.1: Prediction challenges (Thakkar and Bhavsar 2023)

- Accuracy: The model should forecast values accurately, enabling the CSP to allocate the resources optimally.

- Prediction pattern length: It takes many iterations to decide the exact length of predictions. However, it sometimes depends on the nature of the applications.

2.5 Research gaps and findings

This chapter presented various research works discussing elasticity in the context of DSP systems and multiple processing levels in a cloud environment. The forecasting techniques for attaining resource elasticity in a cloud environment are also discussed. Though the research performed in the field of cloud resource utilization is encouraging,

there is scope for improvement, and hence, this work aims to address various research gaps listed as follows:

- In the literature, many elasticity approaches are proposed; however, most of them are designed for processing specific applications (Marangozova-Martin, De Palma, and El Rheddane 2019; Russo, Cardellini, and Presti 2019). However, any kind of application could be deployed on the cloud. Thus, there is scope for designing a holistic approach that will be able to process the input load uniformly.

- The resource elasticity is addressed by literature at various processing levels separately (Xu, Peng, and Gupta 2016). However, the relationships among them are not addressed in the majority of the works.

- As real-time stream processing applications demand dynamic processing capabilities, it is required to have a framework that accurately forecasts the requirements of different resources simultaneously by optimally utilizing them and without compromising the performance of the system.

Thus, this work aims to address the stated research gaps and provide an efficient solution for enhancing the performance of DSP systems and optimally utilizing cloud resources. The next chapter presents the methodologies for implementing elasticity at multiple processing levels of cloud architecture.

Chapter 3

Methodologies and dataset for proposed framework

This chapter presents the methodologies for implementing elasticity at multiple processing level of cloud architecture while efficiently operating distributed streaming applications. First, the distributed stream processing platforms, which process real-time streaming applications, are elucidated. Then, the different approaches for accurately forecasting resource consumption for attaining elasticity at the server, node, and operator levels are discussed.

3.1 Distributed stream processing platform

A cloud receives enormous processing requests from distributed platforms such as sensors, satellites, and geographically separated regions. Such streaming requests, solicit an efficient processing platform that can quickly scale up and down the computation resources and parallelize the execution.

The stream processing platforms accommodate computational requests immediately upon their arrival. There are three types of stream processing platforms: batch, real-time streaming and hybrid. The batch processing platforms handle processing

requests collected in batches and then operated on. The real-time stream processing platforms process the data immediately upon arrival, with lower latency. These platforms are the ideal choice for time-critical applications, and one of those requires real-time analytics. The hybrid platforms include the abilities of batch and real-time platforms and are suitable for processing approaches where both platforms are necessary to make decisions. Apache Storm (Foundation 2022d) and Apache Samza (Foundation 2019) are real-time stream processing platforms. Along with these pure stream processing frameworks, there are other platforms that process requests in batches. Apache Hadoop (Foundation 2022b) is batch processing based on the MapReduce programming model, which solves the problem of a large amount of data. Apache Spark (Foundation 2021a) and Apache Flink (Foundation 2022a) are hybrid frameworks. Table 3.1 lists the various distributed stream processing frameworks with their merits.

Table 3.1: Stream processing frameworks (Thakkar and Bhavsar 2022)

Name	**Merits**
Apache Hadoop (Foundation 2022b)	Can handle a substantial volume of data
Apache Storm (Foundation 2022d)	Easy to set up and operate, reliable, fault-tolerant
Apache Samza (Foundation 2019)	Can handle high volume data in small or micro batches
Apache Spark (Foundation 2021a)	Can execute on various platforms and connect to various data sources

Apache Flink (Foundation 2022a)	Processes both bounded and unbounded data orders at in-memory speed
Apache Splunk(Foundation 2021b)	Demonstrate consistent performance in demanding environments
Apache Kafka (Foundation 2022c)	Delivers high throughput and offers scalable resources for processing and storage

Streaming platforms possess a programming model for composing streaming data computation logic, represented in a directed acyclic graph (DAG) of data-parallel operators. Figure 3.1 exhibits the graphical representation of a generalized stream processing system. The source DAG operators receive data from input sources (e.g., sensors, social media, etc.), as highlighted on the left side, whereas downstream operators, named sink operators, receive the output of intermediate DAG operators. The processing takes place between input and sink operators. The processing operator implements logic on them and produces the output stream. In the case of input stream fluctuation, the number of operators needs to be updated on the fly.

From the merits listed in Table 3.1, it is identified that the Apache Storm is the best solution for attaining the objective of this work, as it provides a reliable, fault-tolerant, and easy-to-set-up platform for processing real-time streaming applications. By provisioning its scalable architecture, the MeghMesa framework is designed to efficiently scale the resources at the operator, node, and server levels concurrently for processing real-time applications in a cloud environment. The following section discusses the Apache Storm platform.

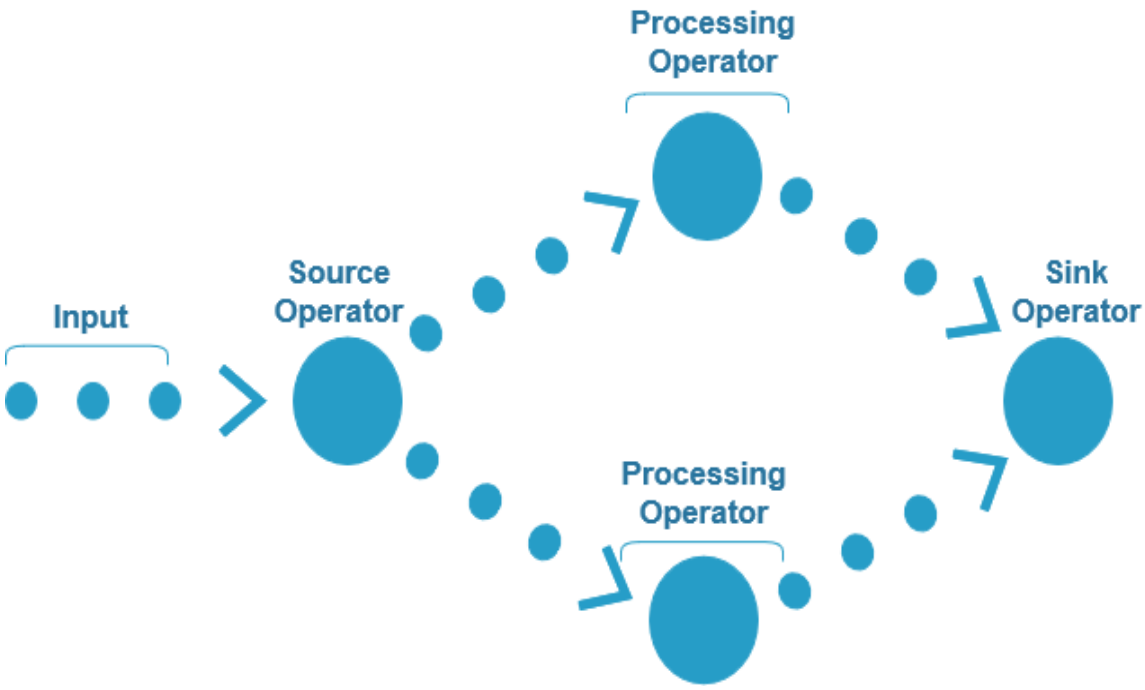


Figure 3.1: Stream processing model (Thakkar and Bhavsar 2022)

3.1.1 A real-time streaming platform

Apache Storm is a distributed and fault-tolerant platform for real-time stream processing (Foundation 2022d). It is an architecture that can swiftly compute complex processes for real-time processing, just like Hadoop does for batch processing. Storm provides a simple programming model and a parallel processing environment for applications. It also allows to perform hot deployment of the system.

A storm application is referred to as topology, a processing logic implemented on data streams with bolts and spouts. It is a directed acyclic graph of spouts and bolts. The spouts are the origin of the data streams, whereas the bolts are processing elements, processing input data streams, and generating valuable output. Figure 3.2 portrays the architecture of apache storm.

The apache storm is a master-slave architecture composed of the following elements (Foundation 2022d):

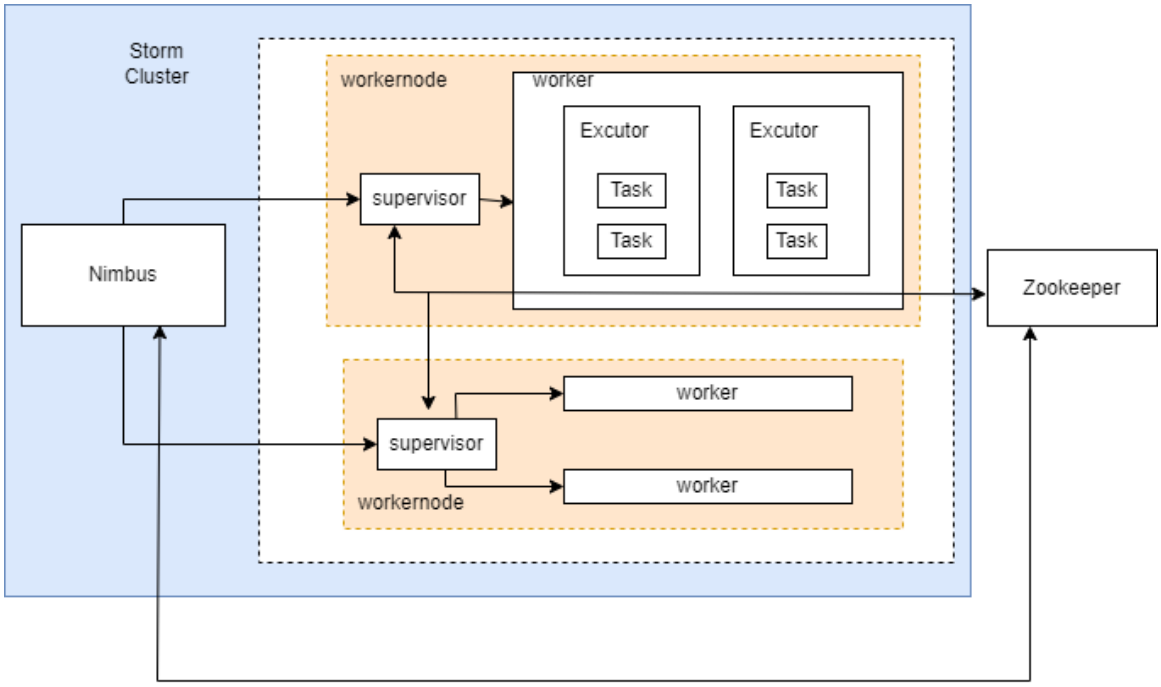


Figure 3.2: Architecture of apache storm (Foundation 2022d)

- Nimbus: a master node responsible for distributing tasks among worker nodes and monitoring the performance.

- Supervisor: a worker node that executes the tasks on workers and coordinates their execution. It is also connected to the master node to get resources and updates on the logic implemented in the worker processes.

- Zookeeper: As apache storm cannot monitor the health and state of the cluster, apache zookeeper is deployed to solve this issue. It is responsible for storing the states of the overall processing executing on apache storm. It brings coordination between the nimbus and supervisors.

3.2 Existing methodologies for attaining elasticity at multilevel

The cloud computing framework allows the execution of numerous applications over the Internet. Such applications demand different amounts of resources for execution. A CSP should deliver an infinite pool of resources to users for the smooth functioning of their applications. Thus, a CSP should have a forecasting model to identify the resource usage pattern. Such a model assists CSP in making resources available for future use. This reduces the scope of service level agreement (SLA) violations between the user and CSP and increases CSP's return on investment (RoI). The optimal resource utilization in multilevel cloud architecture also reduces the carbon footprint in the environment (Thakkar, Trivedi, and Bhavsar 2017).

In the cloud environment, resource utilization is continuously monitored and recorded by the CSP. Time-series data is a set of data that includes a time and one or more values arranged in chronological order. Due to its intrinsic nature as a time series, this type of data may have seasonal and nonseasonal patterns, various trends, missing values, outliers, and complicated relationships between variables. A forecasting model analyzes this time-series data to anticipate resource demand. As application processing demands fluctuate very rapidly, it has always been challenging to accurately predict resource requirements. Statistical methods and conventional neural network-based techniques have been used in existing approaches. Both methods have their advantages and disadvantages; thus, it is essential to carefully consider which approach will work best for the specific situation. Examples of statistics-based classical time series forecasting methods include autoregression (AR) and its other variations, vector autoregressive (VAR), simple exponential smoothing (SES), and others (Hyndman and Athanasopoulos 2018). The conventional neural network models include the recurrent neural network and its variations: long short-

term memory (LSTM) and gated recurrent unit (GRU), artificial neural networks (ANNs) (Borkowski, Schulte, and Hochreiner 2016) and others. Some approaches also used reinforcement learning (Prasad and Bhavsar 2020) to forecast the resource requirements at runtime. By understanding the strengths and weaknesses of each method, one can make a more informed decision and ensure that resource prediction efforts are as effective as possible.

This section discusses the various methodologies and datasets used to design the forecasting model to accurately identify the resource demand at the server, node, and operator levels simultaneously.

3.2.1 Neural Networks

A neural network is a computational model inspired by the structure and functioning of neurons in the brain (Mason et al. 2018). It is also known as an artificial neural network (ANN). The neural network is one of the most prominent fields of machine learning research. It consists of the interconnection of artificial neurons, which are also recognized as units or nodes. All such neurons have weights and biases, are organized in layers, and form a network. ANNs generally comprise an input layer, several hidden layers, and an output layer. The hidden layer is located between the input and output layers. It receives data streams from the input layer, learns the complex representations, and identifies the non-linear relationships from them. The simple block diagram of a neural network shown in Figure 3.3 has three input nodes in the input layer, two hidden layers with four and three neurons each, and one neuron in the output layer. During training, the neural network updates the weights associated with each connection between neurons to accurately predict the output. The backpropagation technique is used to update the network's weight.

A recurrent neural network (RNN) is a type of neural network that processes sequential or time-series data by providing feedback. In a feedforward neural network,

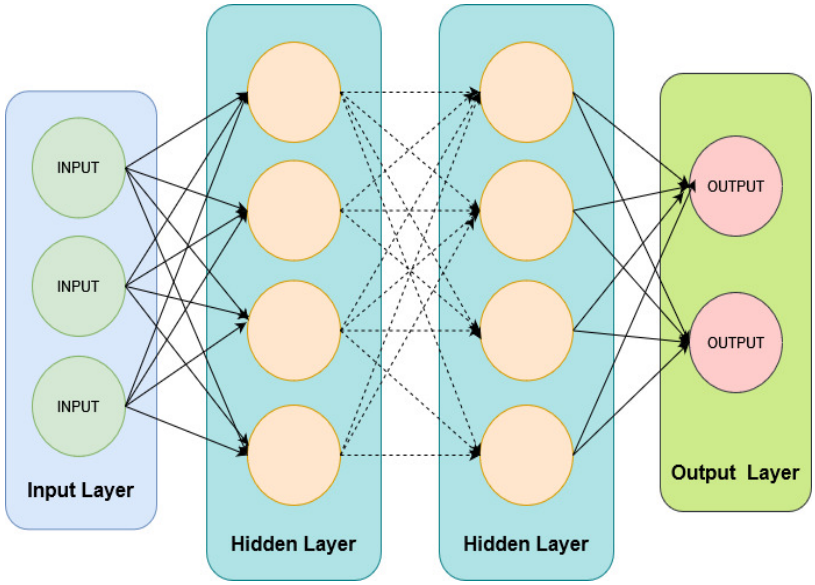


Figure 3.3: A simplified block diagram of neural network (Karim et al. 2021)

there are only forward connections from input to output. However, RNNs have recurrent connections that allow data to be passed from a single step in a sequence. The hidden layer is a principle component in an RNN that maintains an internal state called a "memory" element. At each step in the sequence, RNN takes input, processes it with the previous hidden state, and produces a new hidden state. This process is recurrently iterated for each step in the sequence, and information is updated based on the context of the complete sequence. RNN uses activation functions such as tanh or sigmoid to determine the output of each neuron and the hidden state value at each step. Figure 3.4 illustrates the unrolled sequences of the RNN, where X_t represents the input value at time-stamp t, H_t denotes the hidden state value at time-stamp t, and Y_t represents the output value.

There are two versions of RNN, Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU).

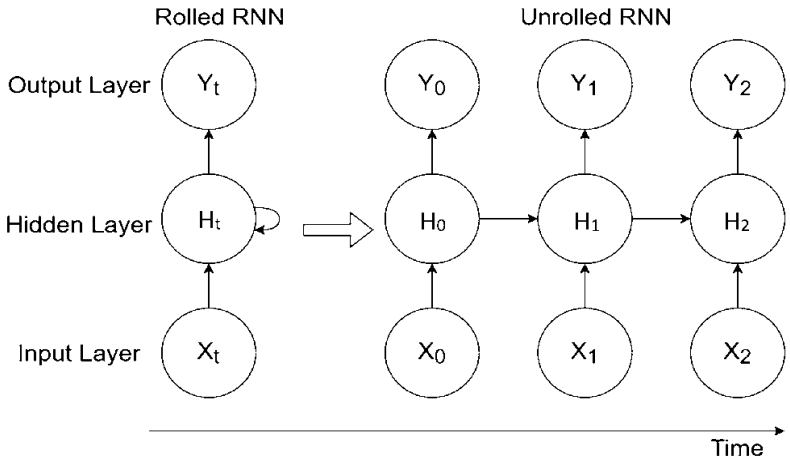


Figure 3.4: Unrolled RNN (Thakkar, Thakkar, and Bhavsar 2023)

3.2.2 Long Short-Term Memory

LSTM is a type of recurrent neural network (RNN) that captures and utilizes long-term dependencies, or "memory", in sequential data, like time series. LSTM networks are designed to address the vanishing gradient problem (back-propagation instability). It captures the long-term dependencies in sequential data. The gradient diminishes parameter updates and becomes negligible over longer sequences in the vanishing gradient problem. LSTM addresses this issue by capturing and propagating critical information over extended periods. The LSTM cell architecture is depicted in Figure 3.5.

A single LSTM cell contains input, output, and forget gates. Each gate includes the activation function, determining the output of neurons based on the input it receives.

Along with the gates, LSTM has the following components (Sherstinsky 2020):

X_t: data input at timestamp t

H_{t-1}: hidden step at timestamp t-1. short-term memory unit

C_{t-1}: cell state at timestamp t-1. memory unit

σ: Activation function

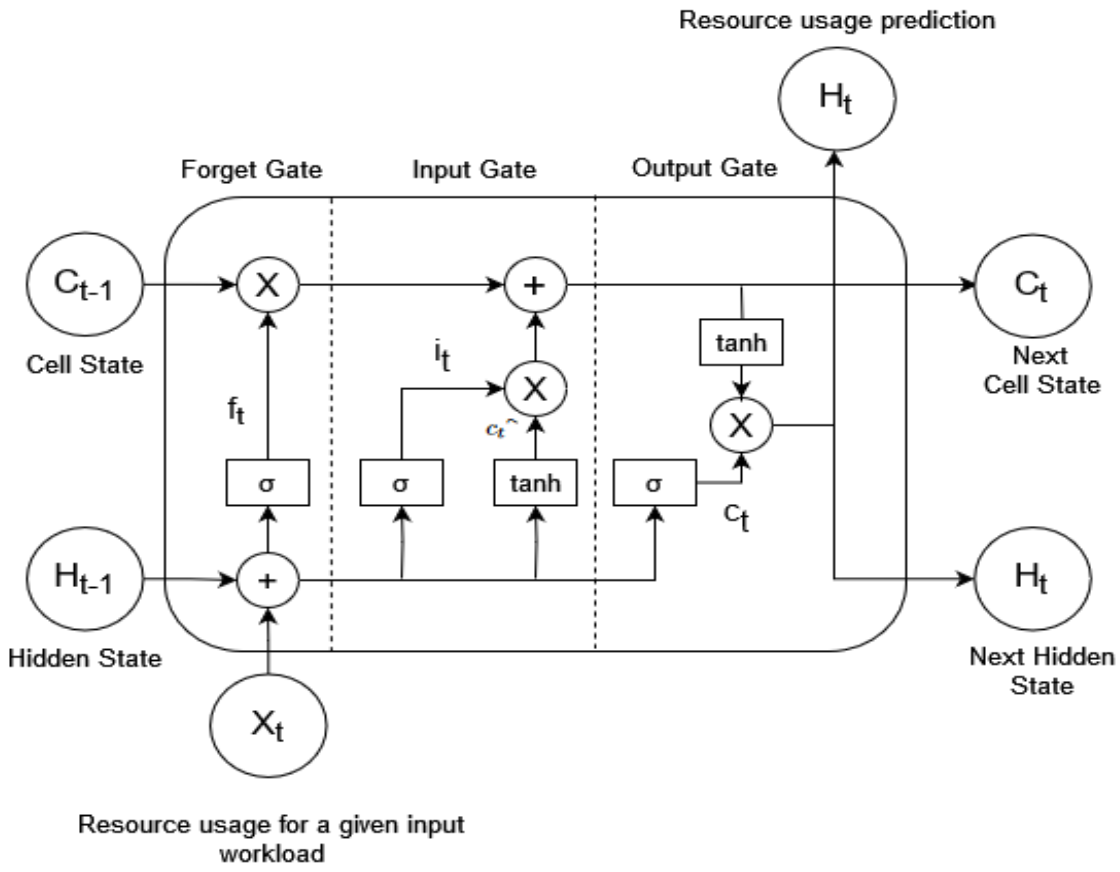


Figure 3.5: LSTM cell architecture (Thakkar, Thakkar, and Bhavsar 2023)

o_t, f_t, i_t: output, forgot, and input gates at timestamp t

C_t: cell state at timestamp t

H_t: output of LSTM cell, input to next hidden state

b_f, b_i, b_o, b_c: bias vectors

Upon receiving a new input timestamp X_t, the LSTM cell proceeds to update its internal gates according to the following sequence (Sherstinsky 2020):

$$f_t = \sigma(W_f[H_{t-1}, X_t] + b_f) \tag{3.1}$$

f_t is the forget gate, which determines whether to remember the output of the previous cell or not. The equation above applies the sigmoid function to the weighted value from the input and hidden state.

$$i_t = \sigma(W_i[H_{t-1}, X_t] + b_i) \tag{3.2}$$

i_t is the input gate, which determines what new information is included in the cell state. In the above equation, the sigmoid function is applied to the current value of variables and the previous hidden state.

$$o_t = \sigma(W_o[H_{t-1}, X_t] + b_o) \tag{3.3}$$

o_t is the output gate, determining what percentage of information from the new cell state passed to the output and hidden state.

$$c_t{}^{\sim} = \tanh(W_c[H_{t-1}, X_t] + b_c) \tag{3.4}$$

The above equation is of the candidate memory cell, resulting from the nonlinear transformation of input values using the activation function.

With the results of internal gates and candidate memory cell state, the next cell state C_t and output H_t are updated with the following formula:

$$C_t = f_t * C_{t-1} + i_t * c_t^{\sim} \tag{3.5}$$

C_t is the state of next cell.

$$H_t = o_t * \tanh(C_t) \tag{3.6}$$

H_t is the output state of the LSTM cell and serves as the successive hidden state.

In the proposed forecasting model, three LSTM units are stacked on top of each other. Through experimental tuning, it was identified that the model with 100 memory cells in each layer performs better. More than 100 cells increased the time and computing complexity, whereas fewer than that were unable to identify the complexity in the data.

3.2.3 Gated Recurrent Unit

A single GRU cell consists of an update and reset gate mechanism for updating the states of hidden neurons (Cho et al. 2014). The update gate controls the flow of information between the previous and current hidden states and determines the amount of previous information to retain and update with new information. The reset gate decides the extent to which the previous hidden state influences the computation of the current hidden state. Each of these gates is connected to the activation function. Figure 3.6 shows the cell architecture of the GRU model.

The GRU cell contains the following components along with gates:

x_t: input vector

h_t: output vector

$\hat{h}_t$: candidate activation vector

z_t: update gate vector

r_t: reset gate vector

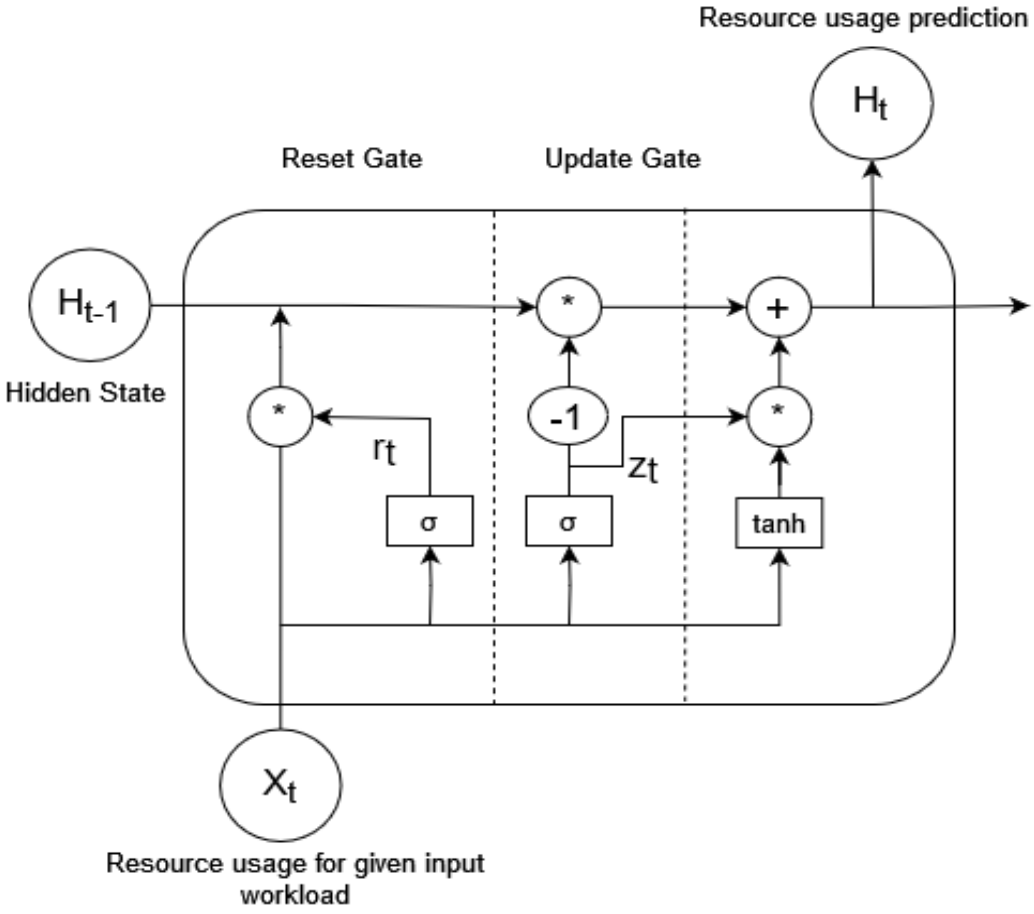


Figure 3.6: GRU cell architecture

σ: activation function

W: weight associated with respected gates

With the given input x_t at timestamp t, the GRU cell updates the internal gate as follows (Cho et al. 2014):

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right) \tag{3.7}$$

The above equation determines the amount of information to forget from the past.

$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right) \tag{3.8}$$

z_t, update gate determines how much past information should be forwarded to the future.

$$\tilde{h}_t = \tanh\left(W_{\tilde{h}} \cdot [r_t \otimes h_{t-1}, x_t]\right) \tag{3.9}$$

With the above formula, relevant information from the past is stored for future processing.

$$h_t = (1 - z_t) \otimes h_{t-1} + z_t \otimes h_t \tag{3.10}$$

The above equation stores information from current memory content and past steps.

$$y_t = \sigma\left(W_o \cdot h_t\right) \tag{3.11}$$

Finally, the above equation forecasts the next value in a sequence.

3.2.4 Statistical analysis models

Statistical models significantly understand the pattern of complex data, and based on that, they perform forecasting. These models consider the mathematical representation of the behaviour of data. Various statistical models include autoregression (AR), moving average (MA), autoregressive moving average (ARMA), autoregres-

sive integrated moving average (ARIMA) and its other variations, vector autoregressive (VAR), Holt-Winters exponential smoothing (HWES), Holt's linear trend method, simple exponential smoothing (SES), and others (Hyndman and Athanasopoulos 2018). This section discusses the autoregressive integrated moving average (ARIMA) and vector autoregressive (VAR) statistical analysis models processed in this work.

Autoregressive Integrated Moving Average (ARIMA)

ARIMA is a regression model that identifies the relationship between the past and present values of parameters. ARIMA can process data collected at regular intervals and is highly dependent on the stationarity of the data. If the data contains any non-stationarity, it should be removed, and the data must be converted to stationary form. For removing trends and periodic events from the time series data, the ARIMA model uses lagged moving averages. ARIMA contains the following components (Hillmer and Tiao 1982):

a. Autoregression (AR): This component refers to the correlation among past and lagged values of the parameter. Here, p indicates the lag order for the given input, which shows the significance of the number of lagged observations on the current observation. In this work, 720 lagged observations were considered for forecasting the value at the present time stamp. Thus, it was AR(720) for the ARIMA model in this work.

b. Integrated (I): This component represents the differences between consecutive values. Thus, it is referred to as d, the order of differencing.

c. Moving average (MA): This component represents the number of lagged observations of errors that significantly affect the current observation. It is referred to as q, the size of the moving average window in ARIMA.

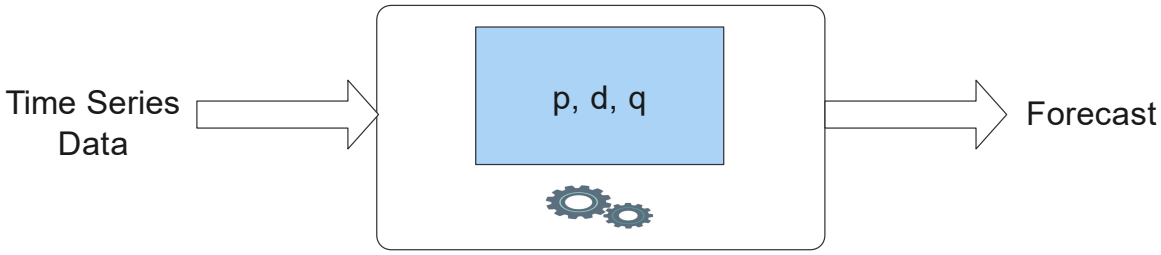


Figure 3.7: Basic ARIMA model (Thakkar and Bhavsar 2023)

Figure 3.7 depicts the basic ARIMA model. The components of ARIMA models are represented as ARIMA(p,d,q). In this work, multilevel real-time streaming data was evaluated using the ARIMA model. The memory requirement for the server level was forecasted by considering 720 past observations.

Vector Autoregressive (VAR) model

The vector autoregression (VAR) model is a statistical model that identifies the relationship between multiple parameters as the change in one or more parameters affects the forecasting result. It is a generalized version of the univariate time series model, ARIMA. The VAR model processes multiple parameters, identifies their relationships, and individually predicts them.

In the first order VAR(1), forecasting each parameter at time t depends on all other parameters at time point t-1, including itself (Haslbeck, Bringmann, and Waldorp 2021). The autocorrelation in VAR(1) shows the effect on the parameter prediction from the previous values of self and cross-lagged impact, which is the prediction of parameters affected by past values of other parameters. The following equation shows the VAR model for p lags (Kotze 2022):

$$\mathbf{y}_t = A_1\mathbf{y}_{t-1} + \ldots + A_p\mathbf{y}_{t-p} + CD_t + \mathbf{u}_t \tag{3.12}$$

where, $\mathbf{y}_t = (y_{1t}, \ldots, y_{kt}, \ldots, y_{Kt})$ for k = 1, ...K

$A_i = (K \times K)$ coefficient matrices for $i = 1, \ldots, p$

$\mathbf{u}_t$ = K-dimensional white noise

C = the coefficient matrix of potentially deterministic regressors with dimension $(K \times M)$ $D_t = (M \times 1)$ column vector holding the appropriate deterministic regressors

In this work, the VAR model was implemented on a multilevel real-time streaming dataset consisting of resource utilization at multiple levels in a cloud environment. For the implementation, resources from the server and node levels were given as input, and requirements for various resources at each level were forecasted.

3.3 Dataset

A dataset refers to a set of data gathered and utilized for training, testing, or evaluating a model. In cloud computing, a dataset can refer to any data collection stored in the cloud. This could include data related to resource usage during business operations, customer transactions, or sensor operations. It may be used for various purposes like resource usage prediction, monitoring, security, and other data analytics.

As this work uses machine learning models to forecast resource utilization, the dataset also plays a crucial role in training and testing the model. The dataset contains dependent and independent parameters that guide the model to recognize hidden patterns in the data. A more extensive and diverse dataset can improve the model's accuracy by providing more representative samples of the data. Thus, two distinct datasets were used in this work to accurately train the model and forecast the resources at the server, node, and operator levels. Based on the forecasting result, a CSP chooses to elastically scale the resources while continuously provisioning services to the users. Hence, the dataset is the fundamental element in the MeghMesa framework.

The following section discusses the bitbrains and multilevel real-time streaming datasets that were used in experiments with MVMS and ML-MVMS forecasting mod-

els.

3.3.1 Bitbrains dataset

The proposed model is evaluated on resource usage data from Bitbrains, a distributed data center that hosts and manages business computations (Shen, Van Beek, and Iosup 2015). There are two tracks in this dataset: fastStorage and Rnd. The fastStorage track contains 1250 VMs, whereas Rnd consists of 500 VMs. Thus, to enhance data diversity, the fastStorage track is selected to assess the performance of the proposed model. The dataset is summarised in Table 3.2. The CSV file for each VM includes its performance metrics, which are listed in Table 3.3.

Table 3.2: Bitbrains dataset summary

Name of the trace	**# VMs**	**Period of Resource usage data collection**	**Interval of Resource usage data collection**	**Total available memory**	**Total available cores**
fastStorage	1,250	1 month	5 minutes	17,729 GB	4,057

As listed in the Table 3.2, the fastStorage dataset contains a total of 1250 VM's resource utilization records, which are collected for the duration of 1 month. According to (Shen, Van Beek, and Iosup 2015), each data point was recorded at an exact interval of 5 minutes. The datacenter had a total memory capacity of 17.720 GB and 4057 CPU cores.

After analyzing each VM file, it was observed that timestamps were not evenly distributed. Therefore, each VM file needs to be pre-proposed. Figure 3.8 shows the plot of unique timestamps in the dataset vs. the different timestamps, representing

the uneven distribution of timestamps in a dataset. Section 3.3.1 describes the procedure followed for the dataset preprocessing. Once the dataset is evenly distributed, it is split into different train:test ratios and evaluated on the proposed model with various combinations of other hyperparameters. The parameters included in the dataset are listed in Table 3.3. Among them, CPU cores and CPU usage parameters were operated to evaluate the performance of the forecasting model.

Table 3.3: Bitbrains dataset parameter

Resource / Parameter name	**Description**
Timestamp	Unix time at data reporting
CPU cores	Number of virtual CPU cores provisioned
CPU capacity provisioned (CPU requested)	Capacity of the CPUs in terms of MHZ, it equals to number of cores x speed per core
CPU usage	In terms of MHZ
CPU usage	In terms of percentage
Memory provisioned (memory requested)	Capacity of the memory of the VM in terms of KB
Memory usage	Amount of memory that is actively used in terms of KB
Disk read throughput	In terms of KB/s
Disk write throughput	In terms of KB/s

Network received throughput	In terms of KB/s
Network transmitted throughput	In terms of KB/s

Experimental setup

The high-processing computing system was preferred as the Bitbrains dataset contained millions of rows. The preprocessing of such data requires a good RAM capacity and computing power to process the data. Thus, the high-processing system with the following configuration was used:

- Operating System: Ubuntu 20.04.6 LTS
- CPU: Intel(R) Xeon(R) Gold 6136 CPU @ 3.00GHz
- RAM: 256 GB

Data pre-processing

The dataset contains memory, network, CPU, and disc read-write usage information of 1250 VMs. As per (Shen, Van Beek, and Iosup 2015), the data is collected at a regular interval of 300 seconds. With the said duration of data collection, there should be 8640 entries for each VM. However, several timestamps are not collected at the interval of 300 seconds. These lagging or leading data entries will induce inefficiency in the model training through erroneous predictions. Also, some VMs contain less than 5000 entries, while others have more than 20,000 entries. Such data entries lead to a different count of timestamps for each VM with a diverse combination. In regards to the original work (Shen, Van Beek, and Iosup 2015), it is required to have 63130 unique timestamps, starting at 1376314846 and ending at 1378906798 UNIX timestamps. With 2591952 seconds and estimating the readings at 300 seconds apart,

there should be $2591952/300 = 8639.84$ timestamps. However, the overall timestamp distribution is uneven, as depicted in Figure 3.8. As missing and redundant entries are recorded at irregular intervals, it requires processing the dataset before further usage.

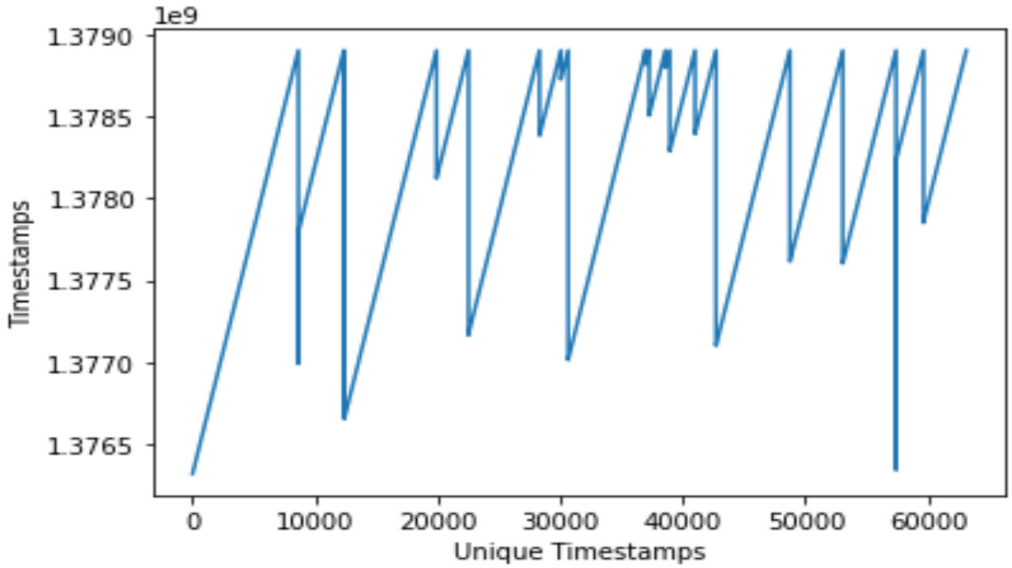


Figure 3.8: Timestamp distribution (Thakkar, Thakkar, and Bhavsar 2023)

The following steps are carried out for pre-processing:

- Data Cleaning
- Data Integration
- Data Normalization

In the first step of data pre-processing, the following iterative equation is used to even the data at time t:

$$X_t = \begin{cases} \sum_{i=0}^{i=T} \sum_{u=max(0,i-s)}^{u=min(T,i+s)} \frac{x_i - x_u}{i-u}, & \text{if } x_i \text{is present} \\ \frac{\int_{i-u}^{i+u} x(v)\, dv\}}{2u}, & \text{if } x_i \text{is not present} \end{cases} \tag{3.13}$$

Here, x is the uneven timestamp, and X is the even timestamp.

The above equation, (3.13), removes duplicate readings and takes samples at an interval of 300 seconds. It returns a smoother and more consistent curve for parameters across all VMs. With a weighted average, it synchronizes the values of all the columns for the missing timestamps found in multiple VMs. As all the columns have been converted to an even distribution of timestamps, the model can now further process them.

After the data is sanitized and integrated for 1250 VMs with 8640 entries each, in the second step, a total of 10800000, all at the synchronized timestamps with even 300-second intervals, are aggregated in a single file.

In the third step, the dataset is normalized for faster convergence and efficient performance in a range of 0-1. The MinMax scalar (Pedregosa et al. 2011) is used for normalization:

$$X\_std = (X - X\_min)/(X\_max - X\_min) \tag{3.14}$$

$$X\_scaled = X\_std * (max - min) + min \tag{3.15}$$

Here, min, max = Feature Range

In the next step, the model is evaluated on the processed data.

3.3.2 Real-time stream processing dataset

As this work is centred around the efficient processing of time-critical applications, this dataset is generated by executing real-time applications in a multilevel cloud environment. The experiment was performed in such a way that the proposed framework receives real-time workloads for processing, where resources requirement were predicted based on the historical usage pattern and current usage is stored in a database for future reference. Different applications were executed on this cluster to introduce

complexity in resource usage patterns.

Numerous platforms are processing distributed streaming applications. Among all of them, Apache Storm (Foundation 2022d) is a distributed and fault-tolerance platform for real-time stream processing. It is a simple programming model and a parallel processing environment for applications that allow for performing hot deployment of the system during runtime. Hence, Apache Storm is the ideal choice for setting up the experiment.

The environment used to deploy and test the proposed approach is established with the following configurations:

Environmental setup

- Operating System: Ubuntu 20.04.5 LTS
- CPU: Intel(R) Xeon(R) CPU E5-1630 v4 @ 3.70GHz
- RAM: 32 GB
- Apache Storm: 20.4.0
- Apache Zookeeper: 3.8.0

Different real-time applications were executed to introduce the model, with real-time complexity in resource usage for the fluctuating workload. Two applications were designed: one was counting the occurrence of the words from real-time data streams, and another was adding the exclamation mark at the end of each word generated in real time. In order to include more diversity in the dataset, the number of applications running at a given point in time kept varying. Such a dataset would become an excellent representative of the problem domain and include data points to guarantee that the forecasting model could learn all the real-time resource consumption patterns without overfitting or underfitting the model. Each application processed infinite data streams per second. These data streams could be in any form, i.e., image,

audio, video, sensor data, business transactions, or live streaming. In this work, we generated text as tuples to be processed by the Storm daemons. The CPU, memory, and other resources at the server, node, and operator levels were recorded every 10 seconds for 40 hours. Thus, the dataset includes 14400 rows, each containing resource consumption for each processing level along with the Unix timestamp. The Unix timestamp represents a time as the number of seconds starting from January 1, 1970, at 00:00:00 UTC. The values for each resource at a given timestamp were recorded in separate CSV files, which were collected in a single CSV file after pre-processing them.

Data pre-prcessing

The memory usage for all the processing levels was recorded in kilobytes, which were converted to megabytes. The other parameters, along with memory and CPU usage, were the number of nodes required at the server level, the number of operators at the node level, and the number of operators consumed at the operator level, recorded into a CSV file. This CSV file was then fed to the proposed forecasting model, ML-MVMS, which learned the hidden resource requirement pattern against the input data streams and accurately predicted the unobserved flow of real-time data streams. Table 3.4 lists and describes each resource recorded at each processing level.

Table 3.4: Dataset configuration

Resource / Parameter name	**Description**	**Processing level**
Timestamp	Unix time at data reporting	All
Threads	Number of processes executed	All

Server Memory	Usage of memory at server level in terms of MB	Server
Sever CPU	Usage of CPU at server level in terms of clock cycle	Server
Number Node	Total nodes in server level	Server
Node Memory	Usage of memory at node level in terms of MB	Node
Node CPU	Usage of CPU at node level in terms of clock cycle	Node
Node total operators	Total operators in node level	Node
Node used operators	User operators in node level	Node
Number of applications	Number of different applications executing at current time	Operator
Application CPU	Amount of CPU allocated to application clock cycle	Operator
Application Memory	Amount of memory allocated to application	Operator
Application req CPU	Amount of CPU requested by application clock cycle	Operator

Application req Memory	Amount of memory requested by application	Operator
Application Task	Number of processes running in operator	Operator

3.4 Evaluation metrics

Evaluation metrics validate the accuracy and effectiveness of the proposed forecasting model and framework and identify the scope of improvement. By comparing the various models using the same error metrics, it is possible to identify the model with the best forecasting accuracy. In this work, we compared the performance of the proposed forecasting model with the other models by comparing the MAE, MSE, and RMSE error metrics. These metrics also provide a direction for improvement while optimizing the model's hyperparameters to improve accuracy. Once the forecasting results are acquired, performance evaluation metrics are used to make the resource scaling decision for the resources at multiple levels of a cloud environment. Such actions improve resource utilization by reducing over- and under-utilization.

The following metrics were used to validate the performance of the forecasting model:

Mean Absolute Error (MAE)

Mean Absolute Error is the mean of the absolute difference between predicted and actual values (James et al. 2013).

$$MAE = (\frac{1}{n}) \sum_{i=1}^{n} |forecast_i - actual_i| \tag{3.16}$$

Mean Squared Error (MSE)

Mean Squared Error is calculated by averaging the squared difference between predicted and actual values. As the errors are squared, MSE only penalizes the absolute deviation from the validation values (James et al. 2013).

$$MSE = (\frac{1}{n}) \sum_{i=1}^{n} (forecast_i - actual_i)^2 \tag{3.17}$$

Root Mean Squared Error (RMSE)

Root Mean Squared Error is calculated by taking the square root of the average of the squared difference between predicted and actual values. RMSE penalizes the outlier, so it is widely used when there are more outliers in data (James et al. 2013).

$$RMSE = \sqrt{(\frac{1}{n}) \sum_{i=1}^{n} (forecast_i - actual_i)^2} \tag{3.18}$$

Mean Absolute Percentage Error (MAPE)

Mean Absolute Percentage Error is calculated by taking an average of the absolute percentage error of each time minus actual values divided by actual values (James et al. 2013). As this metric provides the error measure in percentage, it is easy to use.

$$MAPE = \left(\frac{1}{n}\right) \cdot \sum \left(\frac{|\text{forecast} - \text{actual}|}{|\text{actual}|}\right) \cdot 100 \tag{3.19}$$

For evaluating the performance of the proposed dynamically scaled multilevel framework, the following metrics were used:

Response time

It represents the time taken to process the computing requests. In this work, response time is calculated as the time taken by a model to forecast the resource requirements for incoming requests and scale the resources optimally.

Throughput

This is the number of processing requests a framework handles in a given time. We compared the throughput in terms of the number of threads processed by the proposed framework (MeghMesa) in a given time.

Resource utilization

This metric, in general, presents the amount of various resources required to process the given processing requests in the cloud. In this work, we considered CPU and memory at each processing level, along with the number of nodes executing at the server level and the number of operators running at the node level. This metric helps to improve the optimal usage of each resource across the multiple processing levels of a cloud environment.

Availability

This measures the percentage of time that a cloud service is available and functioning correctly. It is an essential metric for evaluating cloud services' reliability and identifying improvement opportunities.

3.5 Summary

This chapter talked about a real-time stream processing platform, as this work is centred around efficiently executing real-time applications on a multilevel cloud architecture. It was followed by the various methodologies used to attain dynamic scaling at the server, node, and operator levels. The next chapter discusses the design and

development of the framework as a solution to achieve elasticity in a multilevel cloud architecture where real-time streaming applications execute.

Chapter 4

Design and development of MeghMesa framework

This chapter includes a detailed discussion of the development of MeghMesa, the Multilevel Elastic framework for efficiently processing Streaming Applications in a cloud environment. In MeghMesa, Megh represents cloud, M stands for multilevel, E for elasticity, and S and A for streaming applications. The MeghMesa is designed to attain dynamic scaling at different computing levels in a cloud environment while processing highly dynamic real-time applications and optimally utilizing the resources. The conceptual diagram demonstrates the birds-eye view of the MeghMesa framework, which is discussed in depth in an architectural view. The subsequent part of the chapter reviews the individual modules of the architecture. Then, the forecasting model with hyperparameter selection for accurately estimating the resource requirement in real-time is discussed.

4.1 Conceptual diagram of MeghMesa framework

This section discusses the logic behind the concept of the MeghMesa framework, designed to attain the objectives. A CSP receives enormous requests in the cloud

environment, including data processing, storage, retrieval, time critical operations, secure transactions, distributed processing, and many more. Each request demands a different share of individual resources, which fluctuates very often. A CSP should provide customers with infinite resources to process their highly dynamic needs. With the aim of providing seamless access to processing, a CSP must organize the resources optimally and quickly to fulfill the customer's computing demands. A CSP can attain this goal by utilizing the resources at each processing level in a cloud architecture's hierarchy. The cloud environment consists of a multilevel processing architecture, where each level contains numerous computing resources. The multilevel cloud architecture includes server, node, and operator levels. The server level is the topmost level, and the operator level is the lowest in the hierarchy. The server level includes heterogeneous physical resources. Inside each of these servers, there are multiple nodes with different configurations. Each node computes several processes, which are executed by operators, the lowest processing level in a cloud environment. The operator can be seen as a black-box processing element that continuously receives incoming data streams, applies a transformation, and generates new outgoing streams as per the application logic. This work considers the CPU and memory resources at each processing level. For the server level, the number of nodes executing on each server is considered, and for the node level, the number of operators is also considered for a multilevel cloud architecture. The main objective of this work is to attain elasticity in all the processing levels of a cloud system to ensure the seamless functioning of real-time streaming applications.

Figure 5.1 exhibits the MeghMesa, the proposed conceptual framework, designed for handling elasticity at each computing level concurrently in a multilevel cloud environment to enhance the responsiveness and performance of real-time streaming applications.

The proposed MeghMesa framework consists of Global Manager, an ML-MVMS

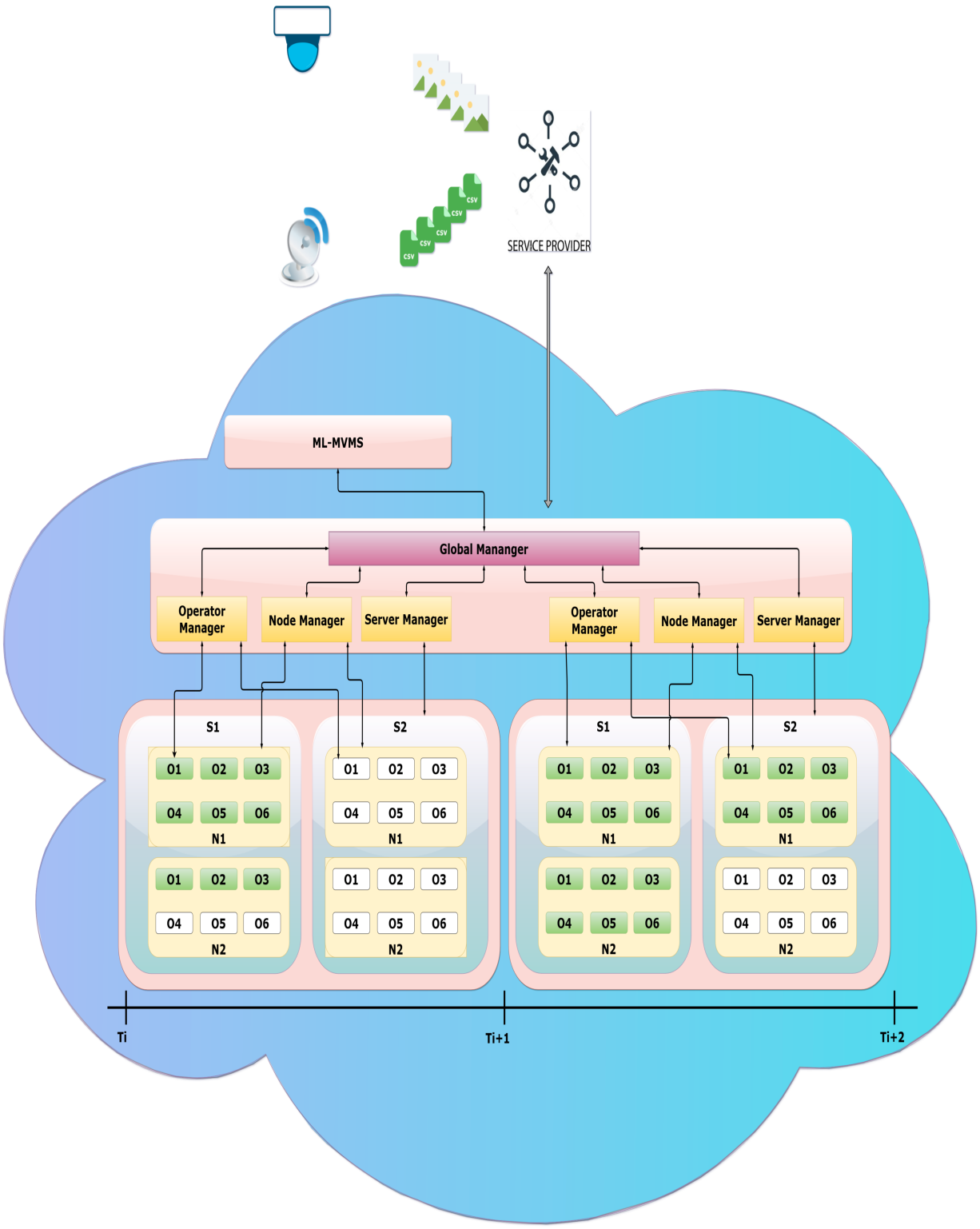


Figure 4.1: Conceptual diagram of MeghMesa (Thakkar and Bhavsar 2022)

forecasting model, a server manager, a node manager, and an operator manager. The Global Manager is a centralized module responsible for efficiently processing real-time streaming applications requests and optimally utilizing the available cloud resources. It receives incoming workloads for different processing purposes, including transforming, analyzing, aggregating, or storing them, from various real-time streaming applications. Such streaming workload demands quick processing; thus, to avail them resources in no time, the Global manager calls the ML-MVMS model. The ML-MVMS model is a prediction module that accurately forecasts resource demand in advance at each processing level in a cloud hierarchy by considering the historical resource usage pattern for given workloads to efficiently process the incoming dynamic workload. It estimates resources required at the server, node, and operator levels for a significant amount of time.

Based on the output of the ML-MVMS and resources configurations at each processing level, the Global manager makes the resource scaling decision at each level. It communicates the scaling decision to the respective level manager. Here, the operator manager is responsible for resources utilization in each operator, denoted as O_i inside node j. The node manager manages the resources usage of each node N_j inside a k^{th} server. The server manager is responsible for resources usage of the server S_k inside a data center. Based on the scaling notification, the respective resource manager performs the scaling operations to accommodate the incoming workload. With the MeghMesa framework, a CSP can efficiently fulfil the fluctuating resource demand of real-time streaming applications and optimally utilize cloud resources.

Figure 5.1 demonstrates how the multilevel elasticity will work over a period of time. During time T_i to T_{i+1}, all operators from node N1 and three operators from node N2 were functioning. However, other operators from nodes N2, N3, and N4 were not utilized. Similarly, the nodes from only server S1 were occupied, and server S2 was in ideal mode. As the processing demand increased from T_{i+1} to T_{i+2}, all the

operators and nodes from server S1 were utilized. Along with that, node N1 from server S2 has also started processing. Thus, as depicted in this figure, based on the incoming processing demands, resources from various processing levels can be scaled in or out to process them efficiently.

4.2 Architecture of MeghMesa framework

The architecture of the MeghMesa framework in Figure 5.2 depicts the complete flow of the proposed work. The Global manager receives processing requests from external sources, including applications demanding quick processing of data, satellite devices sending sensitive information to analyze, surveillance cameras providing live streaming of video, or any baking application performing a number of transactions per second, but not limited to them only.

- In step 1, The Global manager sends the incoming data streams to the scaling decision module, which is responsible for taking scaling decisions at the operator, node, and server levels based on prediction results and resources configuration at each processing level to process the incoming data streams.

- In step 2, the ML-MVMS prediction module considers the historical observations from T_{i-1} to T_{i-n} duration of the data stream history of resources utilization at the operator, node, and server levels for given workloads as a reference to identify the resources demand for upcoming computing requests.

- In step 3, the ML-MVMS module identifies the resources consumption patterns for input processing demand from the historical data and performs the forecasting of the resources demand for a T_{i+1} to T_{i+n} number of timestamps, which allows the CSP to prepare the resources at each processing level well in advance of their requirements. The ML-MVMS module forwards the forecasting values to the scaling decision module.

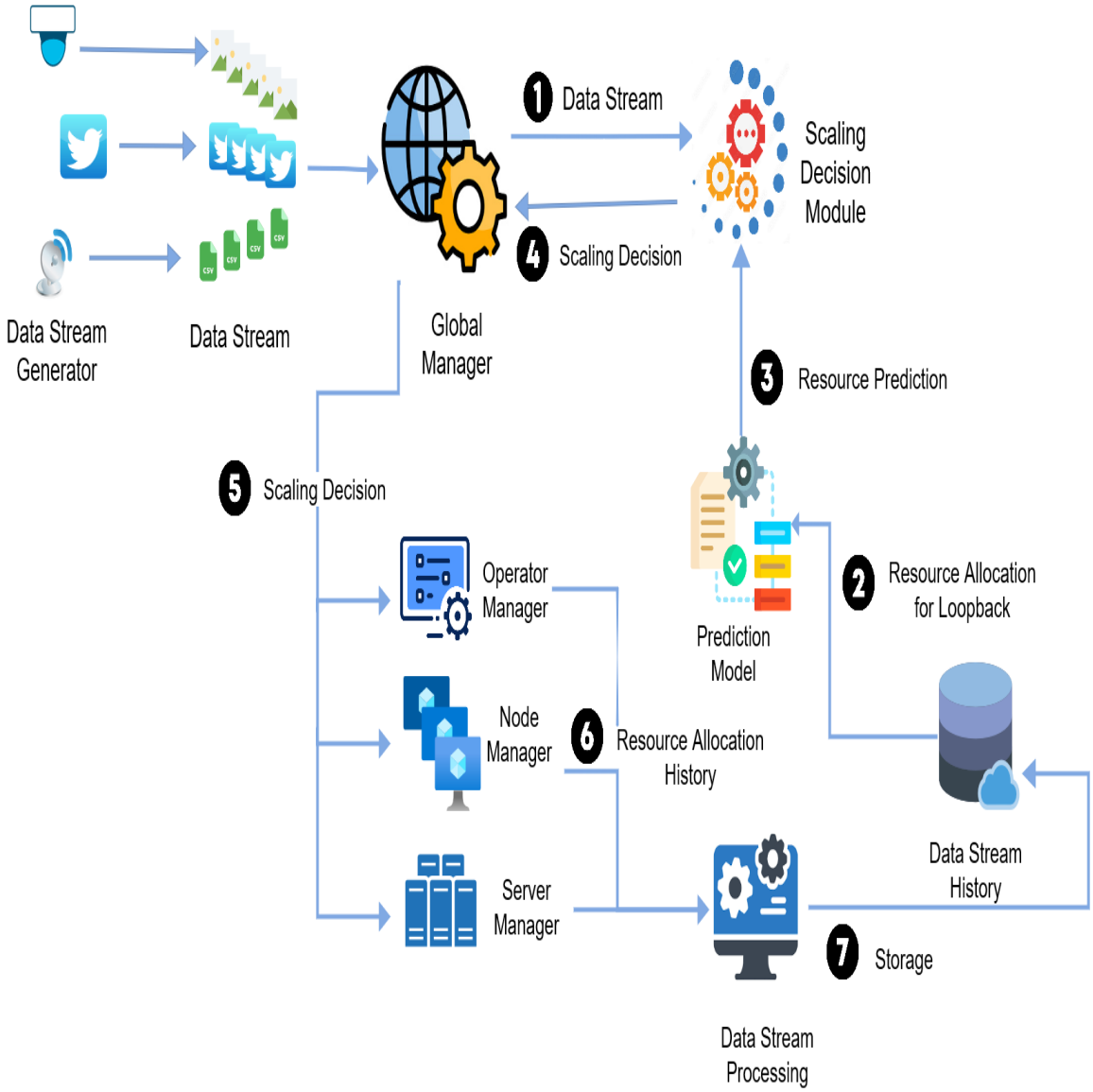


Figure 4.2: Architecture of MeghMesa framework

- In step 4, the scaling decision module checks the feasibility of resources scaling based on resource availability at the processing levels. Considering the maximum capacity of resources for the respective processing level, it decides the scaling of individual resources for each processing level and notifies the Global manager. This module reduces the scope of over- and under-utilization of resources by optimally utilizing their capacity.

- In step 5, the Global manager updates the operator manager, node manager, and server manager about the scaling of resources.

- In step 6, upon receiving resources scaling decision, the respective level manager performs the scaling operations. By quickly scaling the resources, applications can get enough computation infrastructure for processing, and from the CSP perspective, the resources are also optimally utilized.

- After this forecasting and scaling process, in step 7, resources consumption for incoming workload from each processing level is pre-processed and stored for further processing and referencing in step 3. The data pre-processing is performed to transform the raw data into a suitable format for further analysis.

4.3 Design model of MeghMesa framework

This section discusses the design approach of the scaling module and prediction model. Section 5.3.1 discusses the concept behind the decision to scale resources at the server, node, and operator levels. Section 5.3.2 discusses the internal architecture of prediction models: MVMS and ML-MVMS.

4.3.1 Scaling module

This module in the MeghMesa framework takes the decision to scale the resources at respective levels and also determines the amount of each resource type to scale. Upon receiving forecasting values from the prediction module and configurations of individual resources at each processing level from the Global Manager, the scaling module makes the decision to scale resources at each level.

As stated by (Marangozova-Martin, De Palma, and El Rheddane 2019), resources provisioned from the lower processing level improve the overall performance of a cloud. Thus, by referring to that, in this work, the lowest processing level is considered first while taking the scaling decision. The requested CPU and memory are compared at the operator level with the maximum resources available. If the requested resources are available at the operator level, the scaling is only performed at this level, and the operator manager is notified of the scaling; otherwise, the scaling of operators is requested at the node level, and the node manager is informed of the scaling up the resources. The node level, which consists of multiple operators, is on top of the operator level. The scaling module checks the requirements for CPU and memory at the node level by comparing them with the maximum availability. If there are enough resources, then the node manager is notified for the scaling, and in another case, the server manager is reported to scale up the number of nodes. The server level is the topmost level in a cloud hierarchy. After reviewing the availability and scaling necessary at the lower levels, the scaling module examines the CPU, memory, and node availability against the demand. The server manager is notified of the scaling if the resources are available. In the other case, a CSP is requested to increase the resources in a data center. Figure 5.3 shows the conditions to take scaling decisions for extended resource requirements. It can be said that the resources need to be scaled up. When the computational requests are reduced, resources from the higher level are decreased towards, the lower level. Thus, the server level resources are

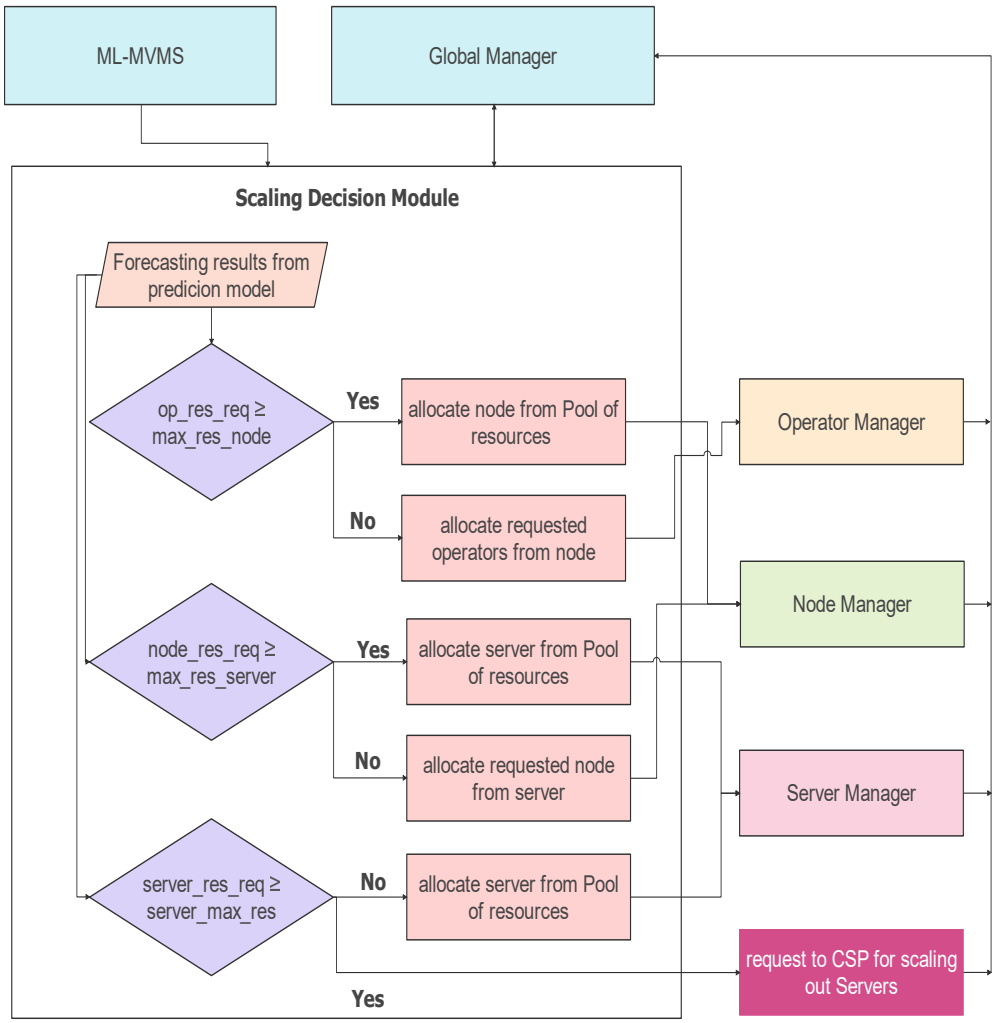


Figure 4.3: Scaling module of MeghMesa framework

consolidated into a few active servers, and others are set to free. (Thakkar, Trivedi, and Bhavsar 2017) proved with their experiment that consolidating the scattered processing elements into fewer reduces power consumption and decreases the carbon footprint in the environment. Subsequently, the nodes are consolidated into a few servers, and the operators are squeezed into the nodes.

4.3.2 Prediction model

In a highly volatile cloud environment, resource allocation must be performed dynamically based on computing needs, which requires the elastic nature of resources for scaling. Resource elasticity can only be effective when resource demand is accurately forecast. Optimized resource allocation necessitates precise resource consumption forecasting, which aids in proactive and real-time decision-making.

The autocorrelation plots unfold the hidden patterns in the data and assist in designing an accurate prediction model. Figure 5.4 shows the autocorrelation plot of the CPU utilization of a VM from the Bitbrains dataset with ten lags, visualizing the long-range dependency. The plot shows a high autocorrelation at lag 0, and then there is the alternate sequence of negative and positive spikes with a negative and positive lag. These lags are due to the change in resource usage for processing real-time workloads. As the demand for resources changes with time, there is a relationship between previous and current resource usage. Thus, LSTM is the best solution due to its inherent capacity to handle long-term dependence on volatile data.

This work is based on the modified LSTM model. The proposed model forecasts the values of multiple resources for multiple time steps in the future. Thus, the proposed resource forecasting model is named the Multi-Variate Multi-Step resource prediction model (MVMS). The MVMS model has evaluated on the bitbrains dataset initially. Then, it is implemented on a real-time multi-level dataset, in which a number of resources at each processing level are forecasted. Thus, the MVMS model for

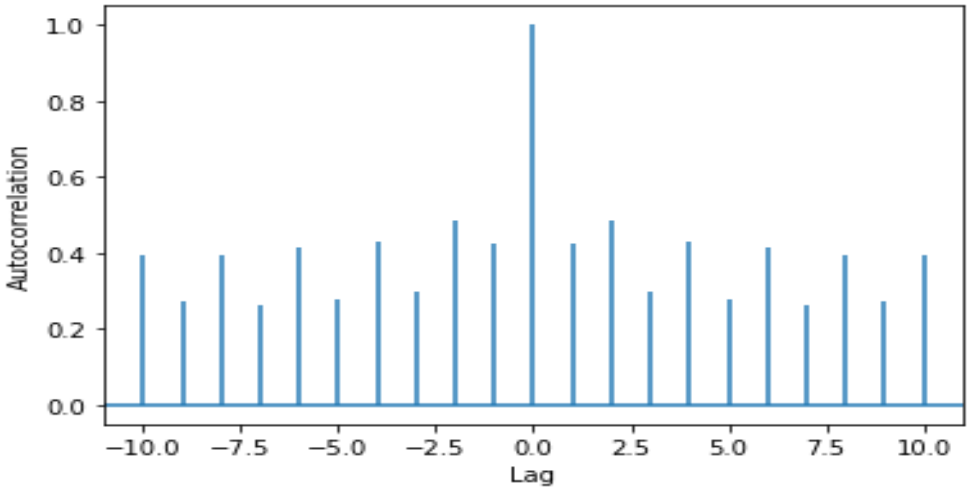


Figure 4.4: Autocorrelation of CPU utilization (Thakkar, Thakkar, and Bhavsar 2023)

prediction in a multi-level environment is referred to as ML-MVMS. The number of input parameters to forecasting models is denominated as i and the number of output parameters as o. Both of these models have the same working prototype and underlying hyperparameters. However, the MVMS and ML-MVMS have varying i and o parameters based on the dataset they are operating over.

Hyper-parameter selection

In this work, the hyperparameters are optimized using a random search method. A random search avoids the grid-related issues of exhaustive methods, making it well-suited for quick exploration of diverse hyperparameter combinations in complex optimization landscapes. Random search is easy to parallelize by concurrently evaluating different random combinations of hyperparameters, which makes it suitable for distributed computing environments where the evaluations are very time-consuming. Table 5.1 lists the hyper-parameters for the proposed forecasting model.

Table 4.1: Hyper-parameters selection for the proposed MVMS model

Hyper-parameter	**Value**
LSTM layers	3
Neurons in each LSTM layer	100
Dense layer	1
Neurons	12
History window	60
Prediction window	12

History and prediction window

The history and prediction windows are the timeframes considered for taking input from the past and forecasting the values for future time durations. The size of the window should be adjusted according to the specific application.

The history window size plays an essential role in determining the usage of resources in a highly dynamic environment. This window includes the number of historical observations of resource usage, and based on the usage pattern for computation requests identified by the model, the demand for future computing resources is forecasted. The prediction window is the time duration for which the forecasting is performed. Thus, the selection of an appropriate history and prediction window plays a significant role while performing the prediction of resources. A suitable prediction window size helps a CSP to scale the resources before their demand, improving resource availability and increasing optimal utilization.

The history window size was derived from work done by (Karim et al. 2021), which has significantly contributed to their optimal output. By referencing the 60 timestamps from history, the forecasting model predicts the resources for 12 timesteps in advance, providing critical information to CSP for preparing resources to cater to input workloads. Figure 5.5 depicts the model's history window and prediction

window selections. Multiple timestep forecasting directs optimal utilization of the cloud data center resources, which increases the return on investment (RoI) for CSP and helps to reduce the carbon footprint (Thakkar, Trivedi, and Bhavsar 2017).

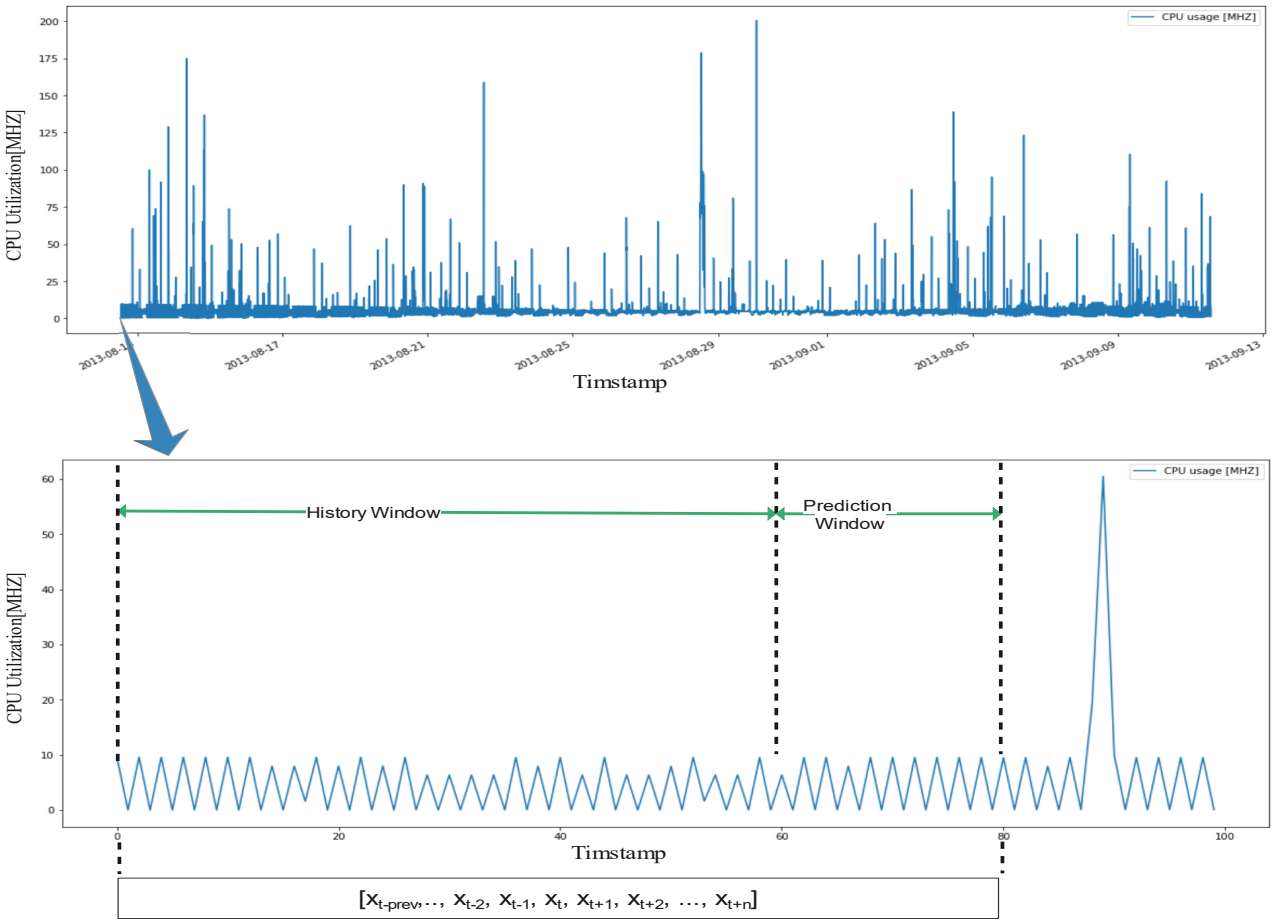


Figure 4.5: History and prediction window (Thakkar, Thakkar, and Bhavsar 2023)

Dataset split analysis

In order to conduct a comprehensive model evaluation and assess the presence of over-fitting or underfitting, the dataset is split into three separate train:test combinations, each represented in a percentage.

- 60:40

- 70:30
- 80:20

In these m:n split ratios, m% of the total dataset is allocated for training, while the remaining n% is allocated for testing.

Optimal batch size

The batch size determines how many training samples are processed before the model changes its parameters based on the error calculated while performing backpropagation. With the smaller batch size, the model takes less data at a point in time, which improves the model's performance. However, it takes a longer time to converge to the result with a smaller batch size. The larger batch size speeds up the training time but deteriorates the result. Thus, the optimal batch size should be considered when training the model for real-time processing applications. In this work, different batch sizes over the prediction model were examined, and by considering the accuracy of forecasting results, the batch size was preferred for designing the resource forecasting model for real-time streaming applications.

4.4 Multi-Variate Multi-Step resource prediction model (MVMS)

The MVMS model is designed based on the LSTM architecture. The LSTM model consists of hidden layers with one or more neurons in each layer. A neuron is a signal processing unit that takes an input signal and uses an activation function to output a signal (Mason et al. 2018). The i input parameters from the Bitbarins dataset were given to the MVMS model, and the value for an o output parameter was predicted. The input parameters were the CPU cores and CPU utilization from historical data, and the output was the CPU usage prediction for the multiple timestamps. By performing parameter sweeps manually and taking an educated guess, it was observed

that a network with three stacked LSTMs having 100 neurons in each layer, followed by a single dense layer with 12 neurons, delivered the optimum performance. The observation revealed that more than three LSTM layers had not improved performance and led to prolonged training time as a consequence of the additional weights to be trained. It was also learned that less than three LSTM layers were not delivering promising performance. The proposed RNN-based model, MVMS, is depicted in Figure 5.6, consisting of an input layer followed by three LSTM layers and a dense layer.

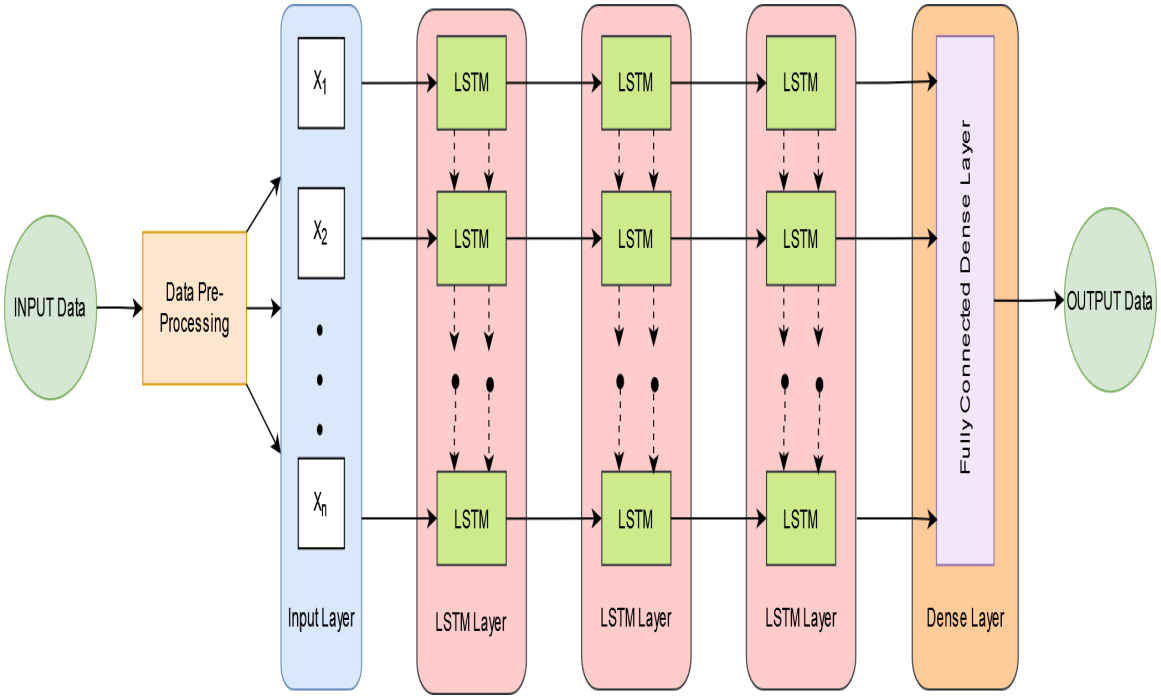


Figure 4.6: The proposed RNN based model (Thakkar, Thakkar, and Bhavsar 2023)

Optimal batch size

In MVMS, for batch size selection, it is essential to subtract the history and prediction window sizes from the total count of timestamps for an accurate multistep forecast. After preprocessing the bitbrains dataset, each VM contains exactly 8640

unique timestamps with a history window of size 60 and a prediction window of 12 steps. Figure 5.5 depicts the history window and prediction window distribution. The history window and prediction window overlap at the current value of the timestamp. Thus, adding one extra value makes the final count of timestamps for each VM 8569, which will be beneficial for prediction. If the batch size is not a perfect divisor of 8569, then in the last batch of the VM, there will be an overlap of data from the next VM, which would cause the model not to comprehend the spike from the VM change and hence reduce the performance. The factors of 8569 are 1, 11, 19, 41, 209, 451, 779, and 8569. The smaller batch size increases the computation time and leads to less information extracted per batch. A batch size value lower than the number of neurons does not contribute more to model convergence. Therefore, factors 1, 11, 19, and 41 were not considered for the batch size, and 209, 451, 779, and 8569 were selected as different batch sizes. Then, with varying combinations of train:test ratio and batch size, the proposed model was iterated 10, 25, and 50 times.

4.5 Multi-Level MVMS (ML-MVMS)

The ML-MVMS model is a multilevel resource forecast version of MVMS that predicts resource demand at the server, node, and operator levels in a cloud environment. The underlying architecture of the ML-MVMS model is identical to the MVMS model. It receives varying values for the i and o parameters from the MVMS model. The model receives CPU and memory usage parameters from the server, node, and operator levels. It also receives node and operator usage from the server and node levels, respectively.

Optimal batch size

The ML-MVMS was operated on the recorded dataset by executing time-critical, complex applications demanding divergent computation power. A smaller batch size

was considered while training the model to track the tiniest change in resource utilization patterns. The data were collected for 40 hours at an interval of 10 seconds. For the real-time dataset, the resource demand prediction was made 1 hour in advance, equal to 360 timestamps. As real-time applications are prone to fluctuations in input workload, more than one hour of prediction may cause resource shortages or wastage while scaling the resources. With the concern of predicting the resource for one hour, the ML-MVMS model was operated over batch sizes of 360 (1 hour) and 720 (2 hours) resource utilization records, respectively, over 100 to 20000 iterations.

4.6 Prediction model execution environment

The prediction models were executed on the Paramshavak, a high performance computing (HPC) system. The Docker image was used to avoid platform dependencies and environmental issues on the Paramshavak system. The Docker image provided high GPU configurations to perform rapid training and testing of the proposed models. The Paramshavak system has the following configurations:

- Operating System: CentOS Linux 7
- CPU: Intel(R) Xeon(R) Gold 6139 CPU @ 2.30GHz
- RAM: 96 GB

4.7 Algorithm for dynamically scaling multilevel resources for processing real-time applications

This section discusses the algorithm developed for attaining elasticity at multiple processing levels in a cloud environment.

Algorithm 1 elucidates the step-by-step procedure for attaining multilevel elasticity for executing real-time streaming applications in a cloud environment. In Step

Algorithm 1 Algorithm for attaining multilevel elasticity

Input: Real-time streaming data to be processed
Output: Scaling decision for processing input
Initialization: Default configuration of the server, node, and operator levels

1: **procedure** MAIN
2: Global Manager ← Real-time streaming data to be processed
3: Forecasting results ← FORECASTING MODEL(resources usage history)
4: Global Manager ← SCALING MODULE(Forecasting results, Default configurations of resources)
5: Sacaling notification to the server, node, and operator manager from Global Manager
6: **end procedure**

1, the Global Manager receives real-time streaming for processing and allocates the resources according to the forecasting. It then calls the forecasting model to estimate the resources required for the subsequent time stamps. In Step 2, the forecasting model delivers the prediction of multiple types of resources at the server, node, and operator levels. In step 3, the scaling decision module decides to dynamically scale each resource at individual processing levels as well as multiple levels concurrently.

Algorithm 2 Algorithm for forecasting resource demand for real-time streaming data

Input: Resource Utilization History
Output: Resource prediction for multistep future demands
Initialization: Input Parameters: i, Output Parameters: o, Hyperparameter configuration

1: **procedure** FORECASTING MODEL(resources usage history)
2: Initialize the hidden state and cell state
3: Update hidden state, cell state, and obtain output
4: Forecast the values for different resources at server, node, and operator levels in a cloud environment for processing real-time applications
5: **end procedure**

Algorithm 2 lists the steps followed for forecasting the demand for multiple resources simultaneously. In step 1, based on the hyperparameter configurations, hidden and cell states were initialized. In step 2, the model was executed for a number of epochs, and in each epoch, the values of the hidden state were updated. In step 3, after the successful execution of the model, the future values of several resources were predicted.

Algorithm 3 presented the logic behind the scaling decision. From steps 1 to 7, it checks for the availability of resources at the operator level, and accordingly, a

Algorithm 3 Algorithm for Scaling of resources

Input: Forecasting results, Default configurations of resources
Output: Scaling decision for server, node, and operator manager
Initialization: Default configuration of the server, node, and operator levels

```
1: procedure Scaling Module(Forecasting results, Default configurations of resources)
2:     if op_res_req ≥ max_res_node then
3:         allocate node from pool of resources
4:         notify Node Manager to scale resources
5:     else
6:         allocate requested operators from node
7:         notify Operator Manager to scale resources
8:     end if
9:     if node_res_req ≥ max_res_server then
10:        allocate server from pool of resources
11:        notify Server Manager to scale resources
12:    else
13:        allocate requested node from server
14:        notify Node Manager to scale resources
15:    end if
16:    if server_res_req ≥ server_max_res then
17:        allocate server from Pool of resources
18:        notify Server Manager to scale resources
19:    else
20:        request to CSP for scaling out server
21:    end if
22: end procedure
```

scaling notification is given to the operator or node manager. Steps 8 to 14 look for the resources at the node level; as a result, the node or server managers are prompted for scaling of resources. At last, resources at the higher processing level are checked for availability; if the server level does not contain enough computing resources, the CSP is informed of the shortage and needs to take administrative actions to provide enough resources to users.

4.8 Summary

This chapter discussed the proposed MeghMesa architecture to attain optimal utilization of resources at runtime while executing time-critical applications in a cloud environment. The forecasting and decision modules play a significant role in dynamically deciding the scaling of resources. The next chapter discusses the performance of the MeghMesa framework and prediction models in various processing environments.

Chapter 5

Implementation, evaluation and performance analysis

The proposed approach was executed and evaluated in three stages:

- Proposed model for achieving dynamic scaling
- Designed the model to attain elasticity at multiple processing levels and experimented on multilevel real-time streaming data for verification of resource optimization and elasticity
- Performance evaluation of the proposed framework

Implementation

The MeghMesa framework comprises a multilevel, multivariable-multistep (ML-MVMS) resource prediction model and a scaling module as primary functional modules. The ML-MVMS accurately identifies resources required at multiple processing levels (server, node, and operator levels) in the cloud environment for a longer duration. Stages 1 and 2 evaluate ML-MVMS to identify its versatility and the scaling decisions made

based on it. Stage 3 evaluates the performance of the MeghMesa framework against existing approaches.

5.1 Stage 1: MVMS: model for achieving dynamic scaling

In stage one, single-level MVMS was evaluated on the Bitbrains dataset. The Figure 5.1 presents the complete sequence to forecast resource usage. The MVMS received CPU utilization in MHz and CPU cores as input parameters and forecasted CPU utilization for fulfilling future demands. The MVMS model learns the CPU utilization pattern from the previous 60 records of CPU utilization and CPU cores and forecasts the CPU utilization for the upcoming 12 timestamps. The outcome of MVMS was validated by evaluating MAE, MSE, and RMSE evaluation metrics.

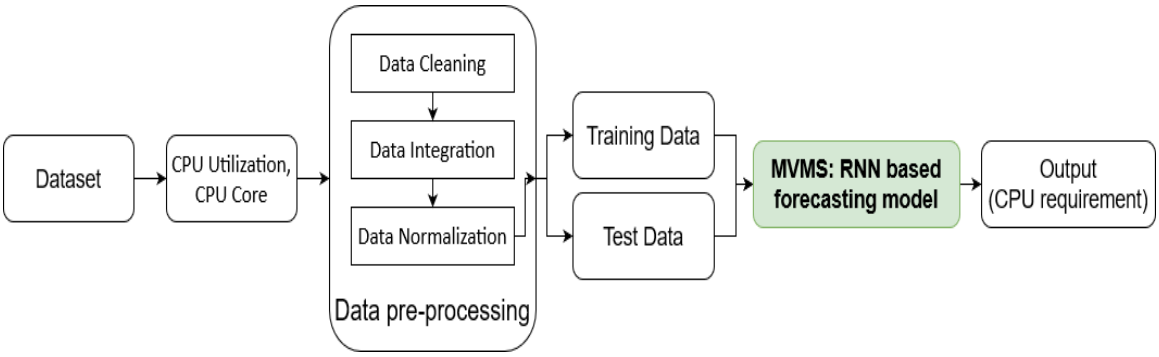


Figure 5.1: MVMS resource usage forecasting sequence

5.1.1 Result analysis

The performance of MVMS was compared with that of the GRU model. The GRU model was designed with the same hyperparameters as the MVMS model, and the results of both models were compared. They were trained over different batch sizes and splitting ratios of a dataset to obtain an accurate forecast.

Both the models were evaluated first on different dataset splitting combinations

and then tested against batch sizes. The models were initially assessed on a 60%:40% dataset splitting ratio and batch size of 8569 over 10 to 50 iterations. The models experimented again with a 70%:30% split ratio over the same hyperparameter configurations. A subtle improvement in performance metrics was observed, with some unfavourable fluctuations. Figure 5.2 shows the average value of MAE for 70%:30% was high as compared to 60%:40%, whereas MSE and RMSE were lower for 70%:30% as depicted in Figures 5.3 and 5.4 for MVMS and GRU.

As the advancement in performance was observed with a higher training ratio, a dataset was further split into an 80%:20% ratio, and models were evaluated with the same hyperparameters. The values of performance parameters in Figures 5.2 to 5.4 concluded that, among all three dataset splitting combinations, models performed optimally with 80%:20%.

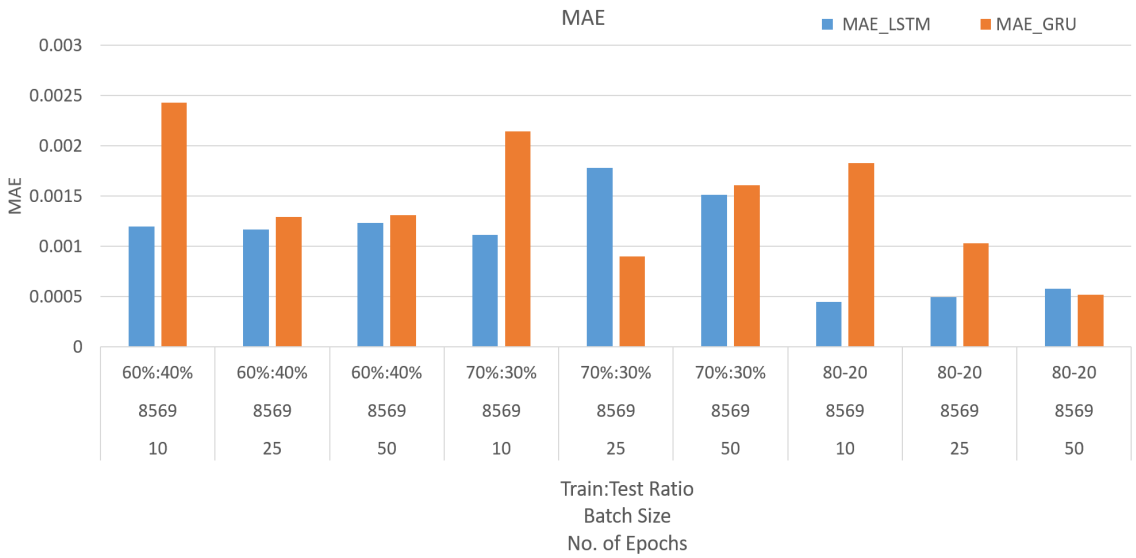


Figure 5.2: Comparison between MAE of MVMS and GRU with different dataset splitting ratios (Thakkar, Thakkar, and Bhavsar 2023)

After identifying the optimal dataset splitting ratio, models experimented with other batches of 779, 451, and 209 sizes. The performance of MVMS and GRU for all

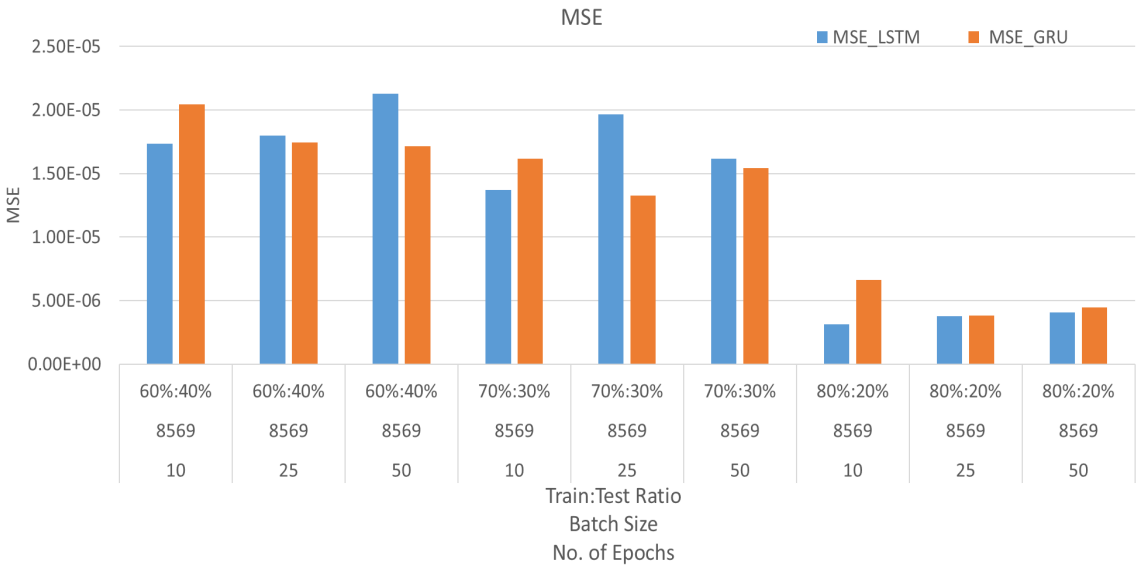


Figure 5.3: Comparison between MSE of MVMS and GRU with different dataset splitting ratios (Thakkar, Thakkar, and Bhavsar 2023)

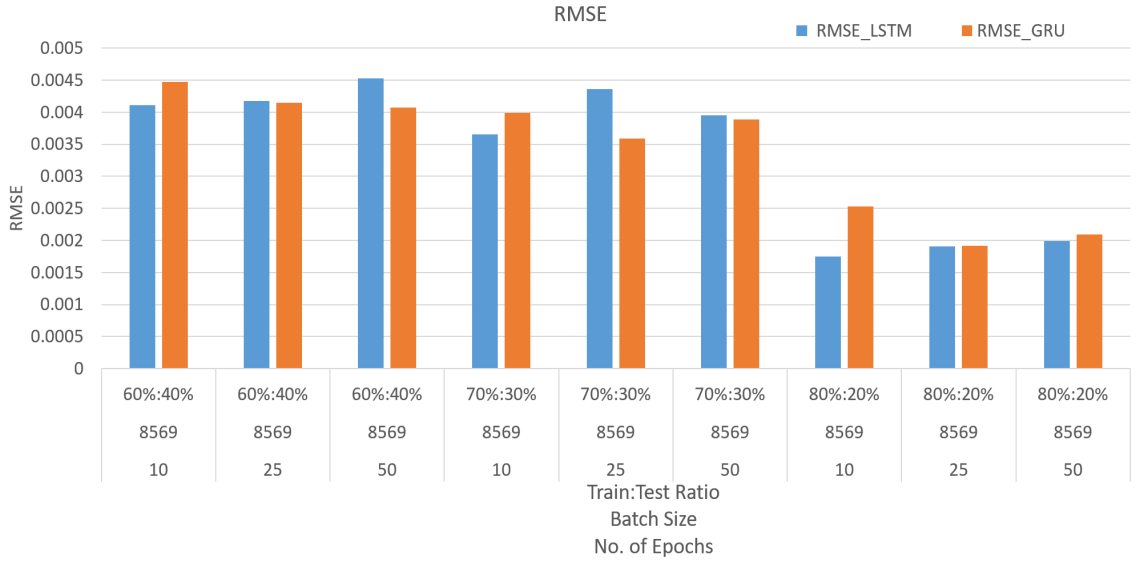


Figure 5.4: Comparison between RMSE of MVMS and GRU with different dataset splitting ratios (Thakkar, Thakkar, and Bhavsar 2023)

batch sizes with an 80%:20% dataset splitting ratio is depicted in Figures 5.5 to 5.7. With the smaller batch size, models were under-fitted, and the performance of both models declined. In contrast to that, with 8569 batch size models optimally performed for 80%:20% dataset splitting ratio. It was observed that with higher training data and larger batch size, models had more data to identify resource utilization patterns. As an outcome, models have accurately forecasted resource requirements. In contrast, with less training data and smaller batch sizes, models failed to identify resource utilization patterns, and eventually, performance decayed.

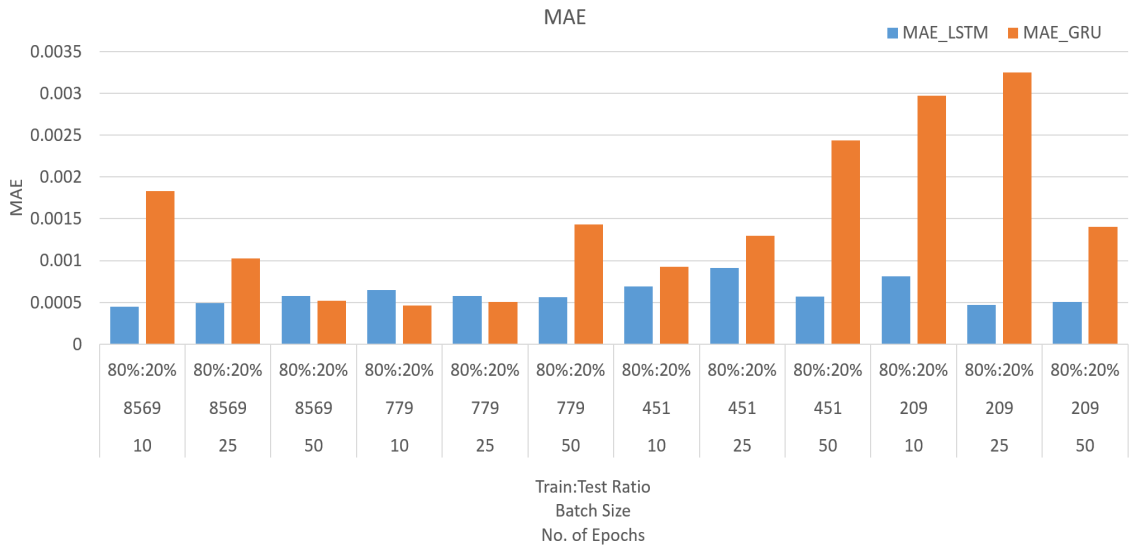


Figure 5.5: Comparison between MAE of MVMS and GRU for all batch sizes (Thakkar, Thakkar, and Bhavsar 2023)

From the performance parameter values of RMSE, MSE, and MAE for the best hyperparameter configuration, it is inferred that the MVMS model outperformed the GRU model. Due to its inherent architecture, the GRU model failed to maintain the values of hidden neurons for longer and fell short of forecasting the CPU requirements with high accuracy. The MVMS model is designed to retain the resource utilization pattern for longer. Thus, it could predict the CPU requirement for multiple times-

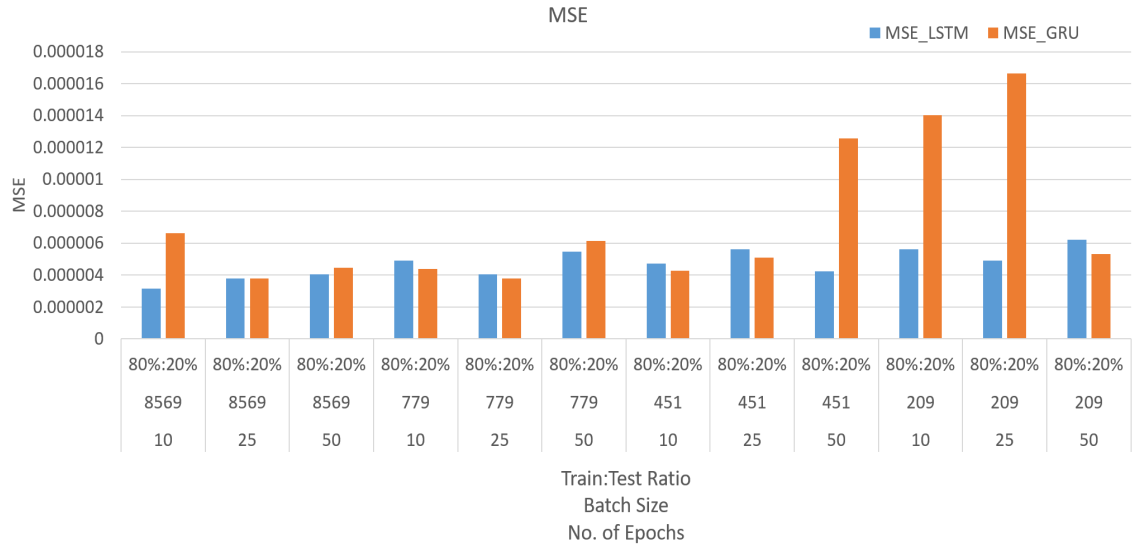


Figure 5.6: Comparison between MSE of MVMS and GRU for all batch sizes (Thakkar, Thakkar, and Bhavsar 2023)

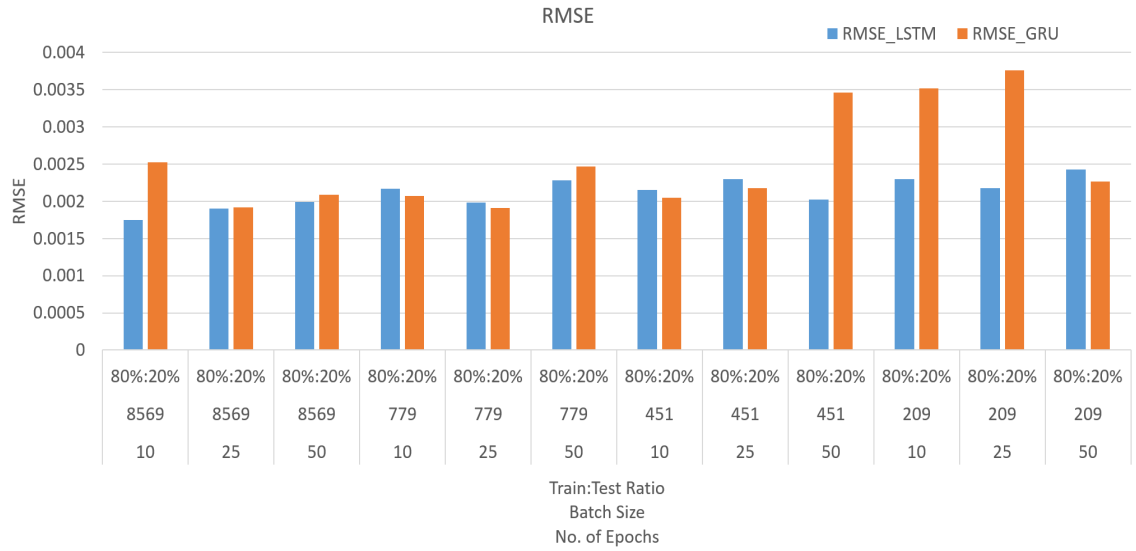


Figure 5.7: Comparison between RMSE of MVMS and GRU for all batch sizes (Thakkar, Thakkar, and Bhavsar 2023)

tamps. The fewer neural network layers in the MVMS model reduce the complexity of the computation and allow for a faster computation time without compromising the accuracy of the result.

5.1.2 Time complexity

The time required by any neural network model to produce the best result depends on the type and number of hyperparameters. Along with that, the time required to train a model also depends on the underlying execution environment. As MVMS was executed on the system with high-end configuration, it took approximately 3 minutes per iteration with the 80%:20% dataset split ratio and 8569 batch size. Thus, within 30 minutes, MVMS converged to the optimal result. However, with a 209 batch size, MVMS took about 12 minutes per iteration. Training times for all the batch sizes with 80%:20% dataset split ratio are shown in Figure 5.8. It was concluded that the model took more time to produce the result for the smaller batch size with any dataset-splitting ratio.

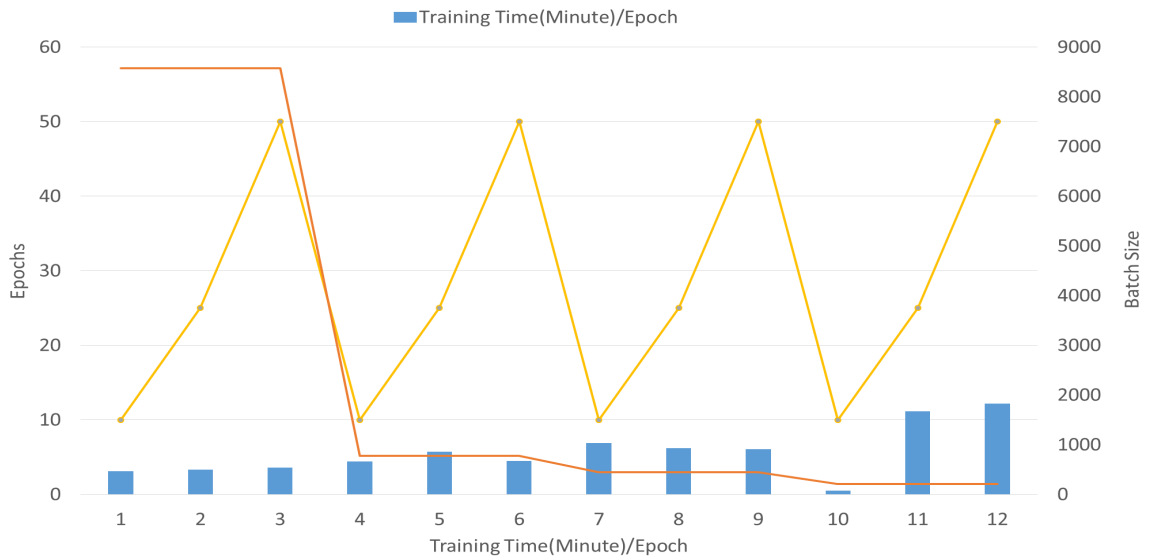


Figure 5.8: Training Time of MVMS (Thakkar, Thakkar, and Bhavsar 2023)

From the results, it is deduced that the proposed neural network architecture (MVMS) for time series prediction is inexpensive to train in terms of processing complexity and time while producing highly accurate results. As the MVMS can predict the resources for individual VMs, it is also referred to as a multi-agent prediction model. Thus, MVMS is versatile enough to work with any problem definition without significant modifications.

As MVMS delivered 99% accuracy, it was implemented on a real-time streaming dataset to accurately identify the fluctuating resource demands in real-time applications and efficiently scale the cloud resources.

5.2 Stage 2: Evaluating the ML-MVMS on multi-level real-time streaming data

The architecture of a cloud consists of a processing hierarchy; at each processing level, several resources are executed. Such resources should be optimally utilized to get the maximum benefit. In this stage, the proposed multilevel MVMS (ML-MVMS) model is proposed to accurately estimate future resource demand; based on that, the resources could be optimally utilized. Efficiently used cloud resources increase return on investment (RoI) and reduce the carbon footprint (Thakkar, Trivedi, and Bhavsar 2017).

The ML-MVMS model was evaluated on real-time streaming data. It retained the utilization pattern of individual resources for a longer time and identified the relationships among the utilization patterns of multiple resources at various processing levels.

5.2.1 Result analysis

This section discusses the performance of MVMS over a multilevel cloud architecture. The multilevel-MVMS (ML-MVMS) receives Timestamp, Threads, Server memory,

Sever CPU, Number of Nodes, Node memory, Node CPU, Node total operators, Node used operators, Number of applications, Application CPU, and Application memory from the real-time streaming dataset as input. By identifying the hidden pattern and relation among all input parameters, the ML-MVMS model predicted the future demand of Server memory, Sever CPU, Number Node, Node memory, Node CPU, Node used operators, Application CPU, and Application memory. The performance of this model was validated by evaluating the MAE, MSE, and RMSE evaluation metrics.

The model was trained with the 80%:20% train:test dataset splitting ratio referenced from the MVMS model. In real-time applications, processing demand changes rapidly; thus, smaller batch sizes were considered in this scenario to track the tiniest change in resource utilization patterns. The model has experimented with 360 and 720 batch sizes over 100 to 20000 epochs.

Figure 5.9 shows the performance parameter values for ML-MVMS executing with a 360 batch size. Initially, the model's performance fluctuated up to 2000 epochs because it was not able to identify the variations in resource utilization patterns. However, the model's performance was continuously enhanced. After 15000 epochs, the model's performance deteriorated due to overfitting of the data. Overfitting occurs as the model starts memorizing the training data instead of learning generalizable patterns. Thus, the ideal time to stop training the model is before it starts overfitting. In this way, as the ML-MVMS with 15000 epochs performed better, it was considered for forecasting real-time streaming applications.

Figure 5.10 displays the forecasted demand of various resources, such as memory, CPU, nodes, and operators, at server, node, and operator levels, respectively, for a 360 batch size. Figures 5.10a to 5.10c shows resurces demand forecasting at server level. Figures 5.10d to 5.10f shows resource demand forecasting at node level. Figures 5.10g and 5.10h shows resources required at the operator level. The performance parameter

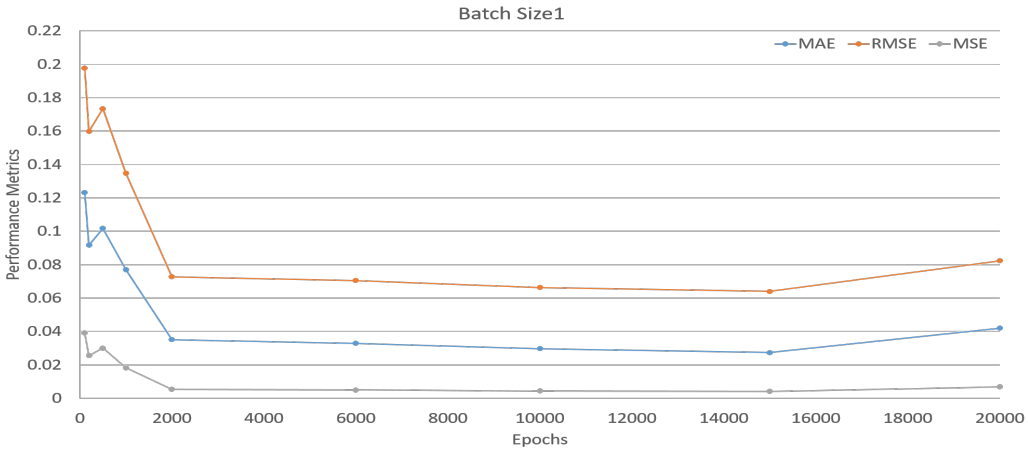
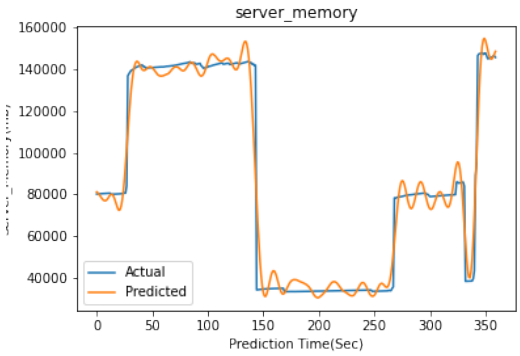


Figure 5.9: Performance of ML-MVMS with 360 batch size

values from Figure 5.9 and Figure 5.10 show that ML-MVMS could identify the resource consumption pattern but failed to accurately predict the resource demand. Thus, the model was experimented with over a 720 batch size.

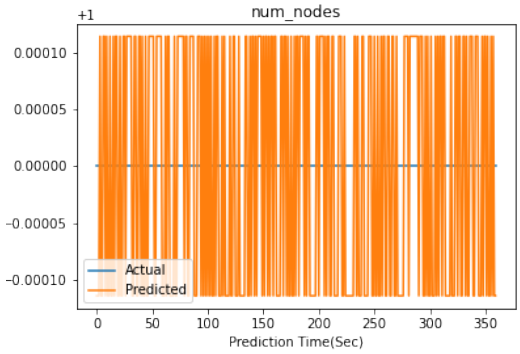
Figure 5.11 shows the performance of the model for 720 batch size. It performed similarly to 360 batch size for the number of epochs. Thus, it was concluded that ML-MVMS performs best for both batch sizes for 15000 epochs.

Figure 5.12 displays the forecasted utilization of memory, CPU, nodes, and operators, at server, node, and operator levels for a 720 batch size. Figures 5.12a to 5.12c shows resurces demand forecasting at server level. Figures 5.12d to 5.12f shows resource demand forecasting at node level. Figures 5.12g and 5.12h shows forecasting of resources required at the operator level. The performance parameter values in Figure 5.11 and Figure 5.12 demonstrated that ML-MVMS accurately identifies the resources needed at each processing level of cloud architecture while processing time-critical applications.

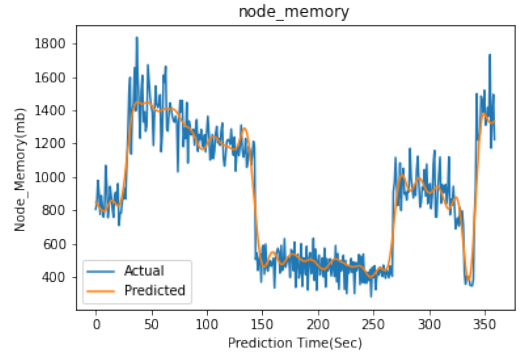


(a) Server Memory(mb)

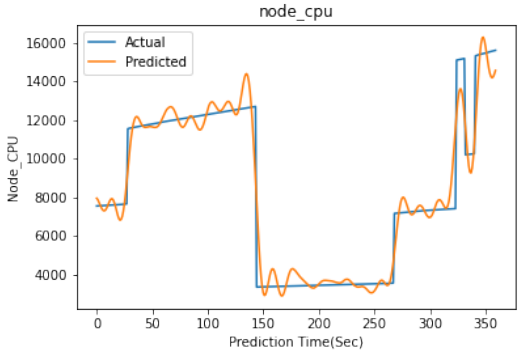
(b) Server CPU



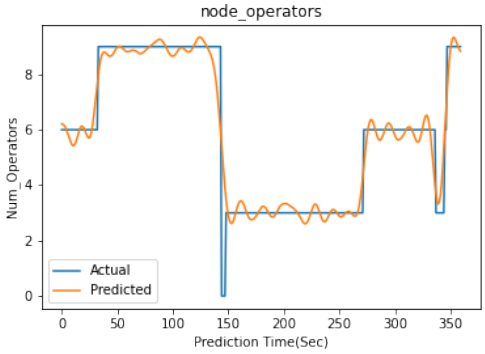
(c) Nodes required



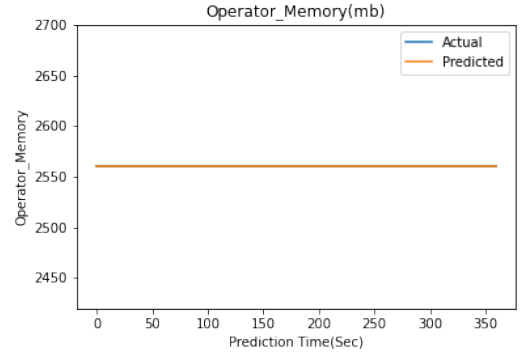
(d) Node Memory(mb)



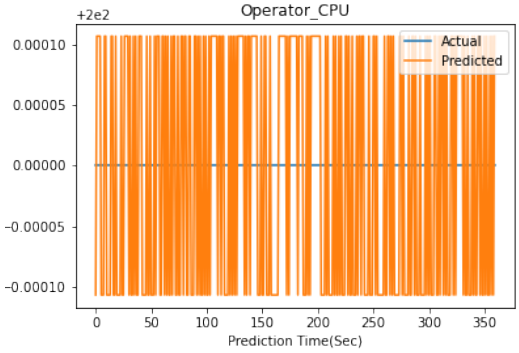
(e) Node CPU



(f) Operator required



(g) Operator Memory(mb)



(h) Operator CPU

Figure 5.10: Resources forecast at Server, Node and Operator level by ML-MVMS over 360 batch size

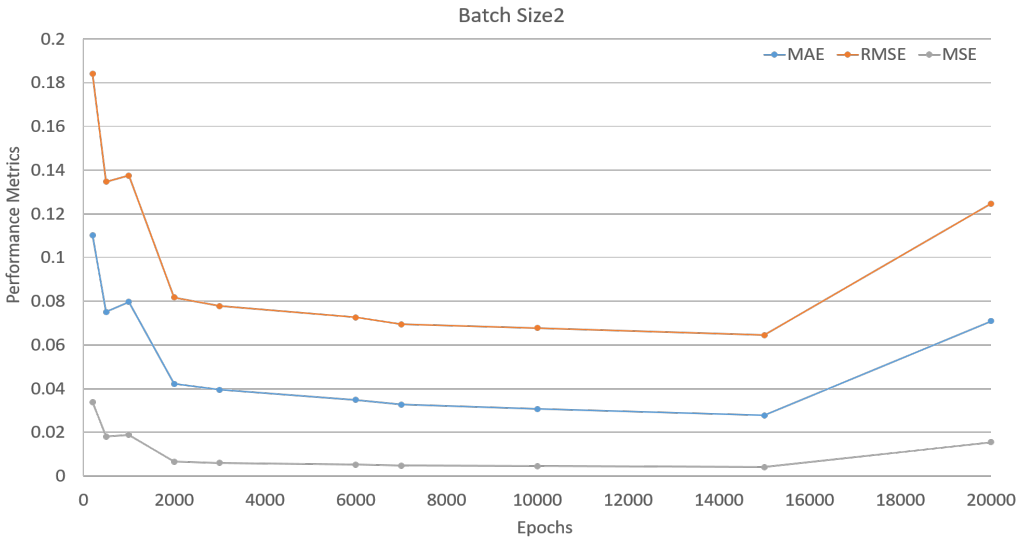


Figure 5.11: Performance of ML-MVMS over 720 bach size

5.2.2 Time complexity

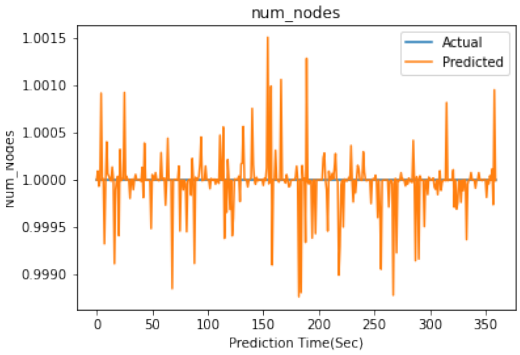
Table 5.1: Time complexity of ML-MVMS

Historical data duration	Prediction data duration	Batch size	Response Time
2 hours	1 hour	360	$\sim$ 16.64 seconds
2 hours	1 hour	720	$\sim$ 5.325 seconds

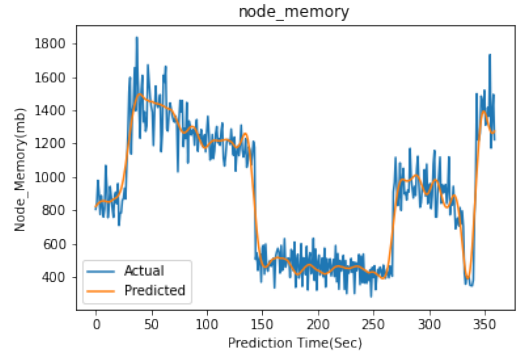
The Table 5.1 shows the time required to execute ML-MVMS for 360 and 720 batch sizes. ML-MVMS takes approximately 16.64 seconds to complete one epoch with a 360 batch size, nearly three times higher than a 720 batch size. ML-MVMS performs better with a 720 batch size in less time, as with a larger batch size, it can process more data and learn the patterns among utilization data. The model requires a higher training time for lower batch size values. Within a more petite

(a) Server Memory(mb)

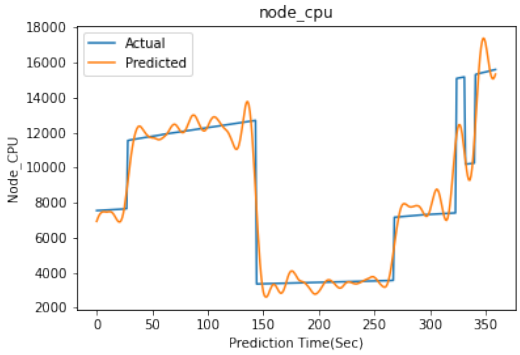
(b) Server CPU



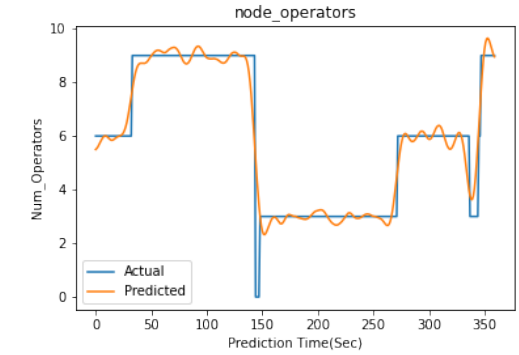
(c) Nodes required



(d) Node Memory(mb)

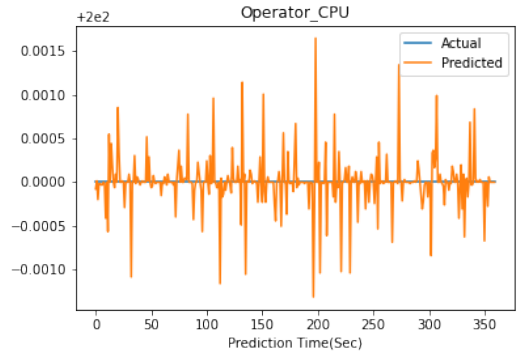


(e) Node CPU



(f) Operator required

(g) Operator Memory(mb)



(h) Operator CPU

Figure 5.12: Resources forecasted at Server, Node and Operator level by ML-MVMS over 720 batch size

time frame of 5.320 seconds, ML-MVMS forecasts resource utilization at the server, node, and operator levels for one hour in advance from a given point in time with the highest accuracy of 0.064554 RMSE, 0.027813571 MAE and 0.004167242 MSE. Here, the lower value of the performance parameter indicates less error in the predicted value compared to the original value. Thus, a given value of RMSE, MAE, and MSE shows that the model delivers highly accurate prediction results. Thus, ML-MVMS with 720 batch size quickly converges to accurate results compared to 360 batch size. With a given forecast time window, CSP can reserve or free resources well before a given time.

In the MeghMesa framework, the ML-MVMS model notifies the Global Manager about the forecasting values for each resource type at multiple processing levels. The Global Manager forwards these values, along with the resource configurations, to the scaling decision module, which is responsible for taking the final decision about scaling each resource type at the individual processing level and among multiple processing levels concurrently.

5.2.3 Statistical analysis on multilevel real-time streaming data

It was observed that the GRU model failed to deliver accurate results for a more extended period. Thus, the performance of the ML-MVMS model was compared with statistical models. Statistical models significantly understand the pattern of complex data, and based on that, they perform forecasting. The ARIMA and VAR models experimented on a multilevel real-time streaming dataset.

Evaluating VAR on multilevel real-time streaming data

As this work addresses multilevel elasticity, it is required to forecast the resources for multiple resources. The vector autoregression (VAR) model is a statistical model that identifies the relationship between multiple resource utilization parameters. It is

a generalized version of the univariate time series model, ARIMA. The VAR model processes multiple parameters together and individually predicts resource utilization parameters. Thus, the VAR statistical model was preferred to evaluate the multilevel real-time streaming dataset.

Table 5.2 lists the input and parameters of the VAR model. Initially, resources from the server and node levels were given as input to the VAR model, and values for server CPU and memory were predicted. The 720 observations from the past were considered, and based on that, values for the next timestamp were estimated. To evaluate the performance of the VAR model, RMSE, MSE, and MAPE error metrics were calculated. The performance parameters in Table 5.2 show that the VAR model delivers very high variation in forecasted values compared to actual resource utilization values for multilevel architecture.

Table 5.2: Performance evaluation of the VAR model

Input Parameters	Server_Memory, Server_CPU, Node_Memory, Node_CPU, Node_Used_Operators
Prediction Parameters	Server_Memory, Server_CPU
Server_CPU	
RMSE	92231.910
MSE	8506725293.108
MAPE	584.74533064992
Server_Memory	
RMSE	1311792.973
MSE	1720800804382.595
MAPE	681.5685939410122

Since more number of parameters were given as input, the VAR model failed to identify the complex resource utilization pattern in the dataset. Thus, the VAR

model was re-evaluated with CPU and memory resources from the server level only. Table 5.3 lists resource utilization parameters from a server level and prediction parameters. It also shows the performance parameters of the VAR model. The result exhibits that the performance of the model deteriorated. Thus, it was concluded that, with its inherent simplicity, the VAR model was unable to ascertain the intricate relationships in resource utilization of a multilevel real-time streaming dataset, as resource requirements at runtime fluctuated very frequently.

Table 5.3: Performance evaluation of the VAR model

Input Parameters	Server_Memory, Server_CPU
Prediction Parameters	Server_Memory, Server_CPU
Server_CPU	
RMSE	299499.157
MSE	89699744789.750
MAPE	1885.4376815910764
Server_Memory	
RMSE	4407451.132
MSE	19425625480527.641
MAPE	2277.5668542990857

To evaluate the performance of statistical models for real-time streaming data, the ARIMA model was experimented on the given input of a resource usage.

Evaluating ARIMA on multilevel real-time streaming data

The autoregressive integrated moving average (ARIMA) model takes the time series of parameters and forecasts future trend. Here, the Server_Memory parameter from a multilevel real-time streaming dataset was given as input to the ARIMA model, which forecasted the value for the same parameter. The 720 lagged observations of server

memory were considered for forecasting the value at the present time stamp. The output of the ARIMA model was evaluated by calculating RMSE, MSE, and MAPE evaluation metrics. Table 5.4 contains the predicted results for Server_Memory. It was derived from the results that the ARIMA model failed to identify the memory utilization pattern at the server level, as ARIMA only considers the autoregression of the server memory utilization values. Since the dataset contains multilevel resource utilization for real-time streaming applications running on the cloud, having fluctuating resource demands over time, the ARIMA model could not identify the complex pattern of resource requirements. Hence, there was a substantial deviation between the forecasted and actual values.

Table 5.4: Performance evaluation of the ARIMA model

Input Parameter	Server_Memory
Prediction Parameter	Server_Memory
RMSE	25592.17116406
MSE	6.54959225e+08
MAPE	15.507375191937184

The real-time streaming data contains seasonal and nonseasonal cycles, different trends, outliers, and complex affinities among the variables. As statistical models heavily rely on the stationarity of data, they can not identify such complexities in the data. ARIMA and VAR models forecast future trends with stationary reference to past trends; therefore, any non-seasons in data will cost the model's performance.

It was observed that the proposed ML-MVMS model could accurately forecast the real-time resource utilization for multiple timestamps in advance for a multilevel cloud architecture, compared to all variations of the VAR and ARIMA models.

5.3 Scaling decision

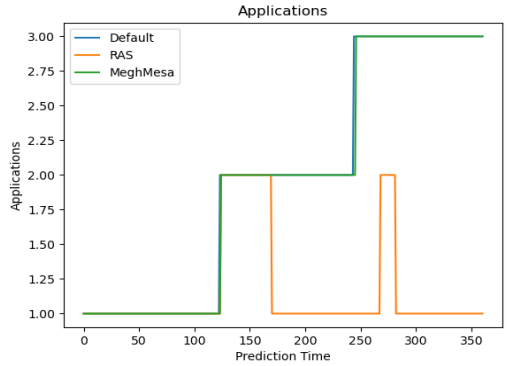
In the MeghMesa framework, the scaling decision module plays an essential role while dynamically scaling the resources. It receives the forecasted values for each resource type and their configurations. The scaling of each type of resource at individual and multiple processing levels was determined based on the availability of the resources. The dynamic scaling decisions will be communicated back to the Global Manager, who will notify the individual processing level managers to scale the resources.

The proposed MeghMesa framework allows CSP to provide high availability of resources for processing real-time applications while reducing resource waste.

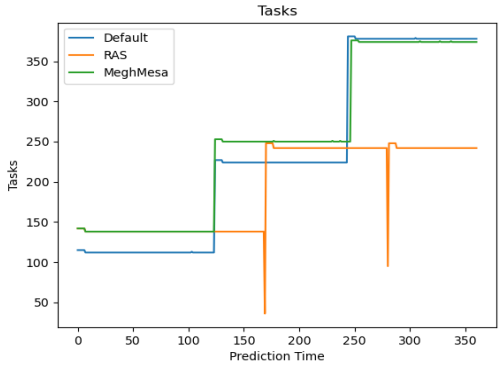
5.4 Stage 3: Performance and result analysis of MeghMesa framework

This section discusses the performance of the MeghMesa framework as compared to the existing approaches for handling elasticity in multilevel cloud architecture. Here, Apache Storm's default resource allocation approach (round robin approach (RR)) and updated resource-aware strategy (RAS) (Peng et al. 2015) were taken as references. Apache Storm's default approach is round-robin, in which a process is evenly distributed among available resources. The RAS works on the principle of improving resource availability and allocating tasks to the most appropriate resources to avoid resource wastage.

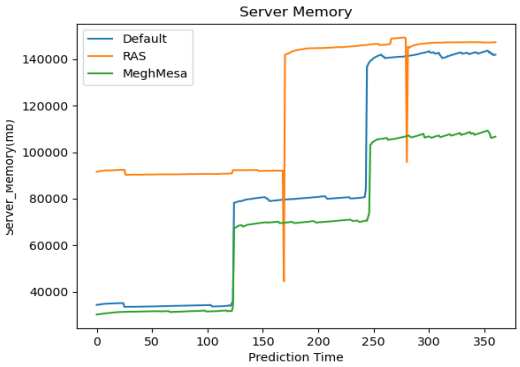
Figure 5.13 shows the comparison of MeghMesa with both of these approaches. The number of applications in Figure 5.13a shows the total applications executing at any given time. As depicted in the figure, RR and MeghMesa can execute the same number of applications; however, RAS cannot run more applications. Figure 5.13b presents the total number of tasks executing for all applications while evaluating each



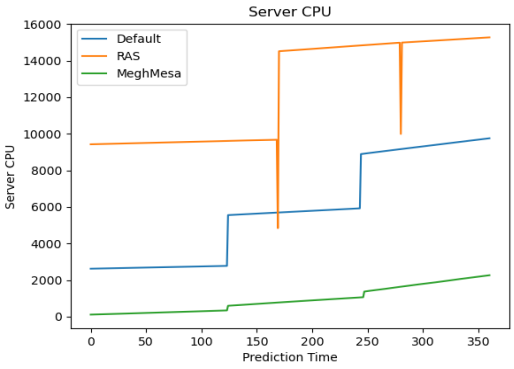
(a) Number of Applications



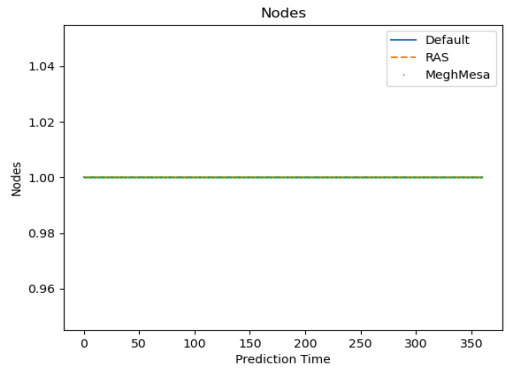
(b) Number of Tasks



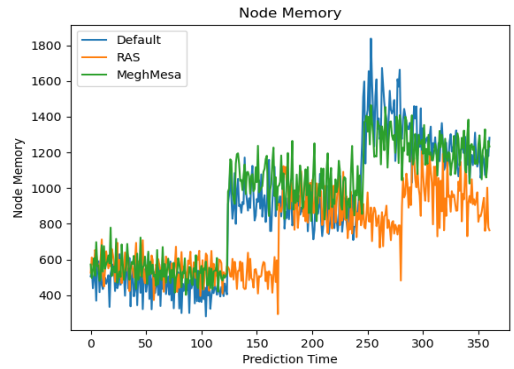
(c) Server Memory(MB)



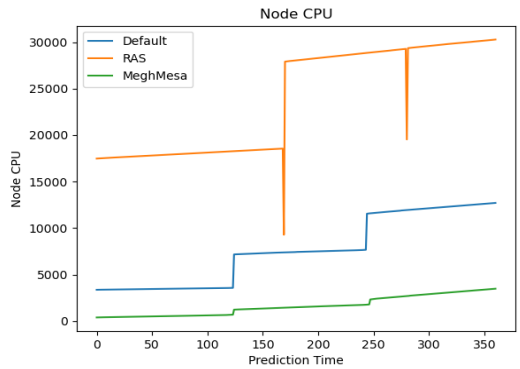
(d) Server CPU



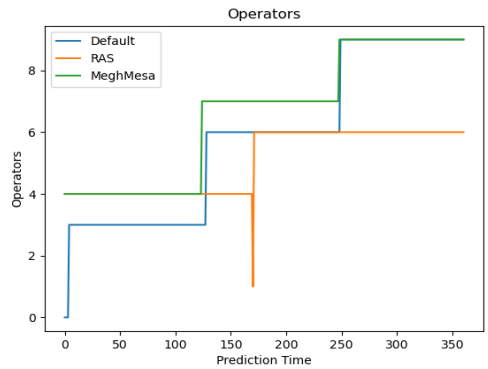
(e) Number of Nodes



(f) Node Memory(MB)



(g) Node CPU



(h) Number of Operators

Figure 5.13: Comparison of MeghMesa with existing approaches

approach. MeghMesa can process the maximum number of tasks at any time as compared to RR and RAS. Figures 5.13c and 5.13d shows the amount of memory and CPU utilized at the server level. The RAS consumes more resources, whereas MeghMesa consumes less while processing additional tasks. Figure 5.13e shows the same amount of nodes required by RR, RAS, and MeghMesa. Resources needed at the node level are presented in Figures 5.13f and 5.13g. At the lower processing level, higher resources are utilized by all of them. However, MeghMesa and RR consume nearly equal amount of memory, but MeghMesa optimally uses CPU. Figure 5.13h plots the number of operators required by each approach. MeghMesa utilized a maximum number of operators at the lowest processing level. The operator level is the lowest in the cloud architecture and costs less than other levels. As per Marangozova-Martin, De Palma, and El Rheddane 2019, resources provisioned from the lower processing level improve performance. Thus, the higher utilization of operators by MeghMesa justifies the better performance compared to RR and RAS. The RR approach overlooks the availability of resources and their demand, leading to resource shortages and sometimes SLA violations. RAS failed to perform better because it continuously checks the resources' availability, leading to a long waiting time, slow processing, and reduced throughput. It also leads to the under and over-utilization of resources.

5.4.1 Performance of MeghMesa framework

Table 5.5 shows the average performance of the MeghMesa as compared to the RR and RAS approaches. The MeghMesa optimally utilized the resources and executed one more application on average for the specific duration of the time, as compared to the RAS approach. However, MeghMesa and RR computed the same number of applications on average. As MeghMesa scaled the resources to handle the incoming workload, it could execute more threads on average compared to the RR and RAS approaches. The operator level in MeghMesa is the lowest processing element; optimal

utilization of it allows the execution of more incoming workload and thus increases the performance of the system. The MeghMesa used additional operators than both approaches and executed more threads into them.

Table 5.5: Performance of MeghMesa

Parameters	**RR**	**RAS**
Application	-0.00831	0.808864
Threads	15.25762	58.23269
Operator	0.761773	1.551247
Node Memory	47.02939	185.2168
Node CPU	-5979.59	-22175.8
Server Memory	-15916.7	-51676.3
Server CPU	-4922.83	-11426.6

Similarly, the amount of memory used was more than the average usage by the RR and RAS approaches. However, MeghMesa took fewer CPU cycles from the node level, but that had not affected the performance of the system. The server is the highest processing level in the MeghMesa framework. Provisioning fewer high-cost processing elements improves the performance of the system. MeghMesa proves this by consuming less server-level resources than RR and RAS approaches.

5.4.2 Space Complexity of MeghMesa framework

The space complexity of the MeghMesa framework is determined by the internal modules: the ML-MVMS and scaling modules. The memory required by the ML-MVMS

model depends on its underlying functional levels, hyperparameter configurations, and the number of input-output parameters. As the ML-MVMS model processes multiple resource utilization parameters from each processing level and predicts resources at each lever, it entails significant memory. However, the scaling module inside the MeghMesa framework leverages a constant amount of memory for making resource scaling decisions at each processing level. Thus, the complexity of the ML-MVMS model introduces more significant memory requirements in the MeghMesa model. Hence, the overall space complexity of the MeghMesa framework is greater than the other two approaches, which take a constant amount of memory to decide the resource scaling.

5.5 Summary

From the results, it is concluded that the MeghMesa framework is able to accurately forecast the resource requirements at multiple processing levels of cloud architecture and dynamically scale them quickly. The time complexity of multilevel MVMS (ML-MVMS) shows that it takes a few seconds to forecast the resources in advance. This leads to the proposed framework handling any fluctuation in resource requirements at runtime well in advance.

The MeghMesa framework benefited CSP by allowing them to highly utilize the resources, generate a high Return on Investment (RoI), and provide customers with uninterrupted resources for processing time-critical applications in real-time.

Chapter 6

Conclusion and Future scope

6.1 Conclusion

- The MeghMesa framework can accurately determine the resources required at the server, node, and operator level in a cloud environment.

- The MeghMesa quickly converges to optimal results while processing real-time streaming data.

- The proposed framework comprises an ML-MVMS resource prediction module that accurately predicts resource demands. The efficiency of the model depends on the selection of hyperparameters inside the model. The optimal combination of hyperparameters determines the accuracy of the forecasting model. In the ML-MVMS model, the hyperparameters are optimized by experimentally tuning them with a random search technique. Random search is easy to parallelize by concurrently evaluating different random combinations of hyperparameters, which makes it suitable for distributed computing environments where the evaluations are very time-consuming.

- From the performance evaluation, it is derived that the MeghMesa framework

outperforms the existing approaches by optimally utilizing resources and quickly availing resources on demand.

6.2 Furture scope

- Implementing the proposed framework on a working environment.
- The reinforcement learning approach can be used to take scaling decisions after forecasting resource usage. This allows the system to take the scaling decision based on the workload at the current time.
- In this work, the proposed multilevel elasticity is attained in the context of apache storm. However, it can be evaluated on public cloud platforms or other stream processing platforms.
- The MeghMesa framework can be easily customized and extended for different requirements, as it is general and flexible.

In ML-MVMS, resources at multilevel were forecasted for one hour in advance from a given point in time; however, depending on the demand of CSP, this time frame can be varied.

Here, relationships among resource utilization parameters were manually determined for prediction; however, various resources at multiple levels can be dynamically identified using machine learning approaches.

6.3 Real-life applications of the MeghMesa framework

The MeghMesa framework facilitates the CSP to host real-time stream processing applications, demanding resources in an Ebb and Flow pattern. It precisely identifies the resource demand of different applications and quickly allocates the optimal

amount of resources from each computing level of the cloud hierarchy. This way, with the help of MeghMesa, the CSP can optimally utilize the resources while efficiently catering to cloud consumers.

As the MeghMesa framework is easy to adapt, the private cloud owner can utilize it to govern the resources optimally. It allows the execution of resource-intensive processes in parallel by elastically scaling resources. Such processes include satellite data processing, medical computations, surveillance applications, social media applications, and scientific and research-related processes.

The MeghMesa framework identifies the demand for resources based on their utilization pattern, which facilitates the CSP in planning resource availability accordingly.

The real-time applications that provide livestock price tracking are time- and money-sensitive. It requires quick changes in prices and a total number of shares for its users in real-time. The MeghMesa framework identifies the flow of a number of users accessing this platform based on regular usage and provides them with the resources to process the stock price query and their transactions in real time. This framework can also quickly process the sudden demand for processing by elastically scaling the resources.

Publication details

- Thakkar, Riddhi , Dhyan Thakkar, and Madhuri Bhavsar. "MVMS: RNN based Pro-Active Resource Scaling in Cloud Environment." Scalable Computing: Practice and Experience 24.1 (2023): 17-33. (SCOPUS Indexed, Web of Science)

- Thakkar, Riddhi, and Madhuri Bhavsar. "Achieving multilevel elasticity for distributed stream processing systems in the cloud environment: A review and conceptual framework." Proceedings of the 2022 Fourteenth International Conference on Contemporary Computing. 2022. (SCOPUS Indexed, Included in ACM Digital Library)

- Thakkar, Riddhi, and Madhuri Bhavsar. "NAARPreC: A Novel Approach for Adaptive Resource Prediction in Cloud." Soft Computing and Its Engineering Applications: 4th International Conference, icSoftComp 2022, Changa, Anand, India, December 9–10, 2022, Proceedings. Cham: Springer Nature Switzerland, 2023. (SCOPUS Indexed)

Bibliography

Assuncao, Marcos Dias de, Alexandre da Silva Veith, and Rajkumar Buyya (2018). "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions." In: *Journal of Network and Computer Applications* 103, pp. 1–17.

Bhatia, Jitendra et al. (2019). "Software defined vehicular networks: A comprehensive review." In: *International Journal of Communication Systems* 32.12, e4005.

Bibal Benifa, JV and D Dejey (2019). "Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment." In: *Mobile Networks and Applications* 24, pp. 1348–1363.

Borkowski, Michael, Christoph Hochreiner, and Stefan Schulte (2019). "Minimizing cost by reducing scaling operations in distributed stream processing." In: *Proceedings of the VLDB Endowment* 12.7, pp. 724–737.

Borkowski, Michael, Stefan Schulte, and Christoph Hochreiner (2016). "Predicting cloud resource utilization." In: *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pp. 37–42.

Calheiros, Rodrigo N et al. (2011). "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms." In: *Software: Practice and experience* 41.1, pp. 23–50.

Cardellini, Valeria et al. (2018). "Optimal operator deployment and replication for elastic distributed data stream processing." In: *Concurrency and Computation: Practice and Experience* 30.9, e4334.

Chen, Hongjie et al. (2021). "Graph Deep Factors for Forecasting with Applications to Cloud Resource Allocation." In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pp. 106–116.

Cho, Kyunghyun et al. (2014). "Learning phrase representations using RNN encoder-decoder for statistical machine translation." In: *arXiv preprint arXiv:1406.1078*.

Chudasama, Vipul and Madhuri Bhavsar (2020). "A dynamic prediction for elastic resource allocation in hybrid cloud environment." In: *Scalable Computing: Practice and Experience* 21.4, pp. 661–672.

Dinda, Peter A (1996). *The statistical properties of host load (extended version)*.

Eskandari, Leila et al. (2018). "T3-Scheduler: A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster." In: *Future Generation Computer Systems* 89, pp. 617–632.

— (2021). "I-Scheduler: Iterative scheduling for distributed stream processing systems." In: *Future Generation Computer Systems* 117, pp. 219–233.

Farrokh, Mohammadreza et al. (2022). "SP-ant: An ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters." In: *Expert Systems with Applications* 191, p. 116322.

Foundation, Apache Software (2019). *Apache Samza*. URL: http://samza.apache.org/.

— (2021a). *Apache Spark*. URL: https://spark.apache.org/.

— (2021b). *Apache Splunk*. URL: https://www.splunk.com/.

— (2022a). *Apache Flink*. URL: https://flink.apache.org/.

— (2022b). *Apache Hadoop*. URL: https://hadoop.apache.org/.

— (2022c). *Apache Kafka*. URL: https://kafka.apache.org/.

— (2022d). *Apache Storm*. URL: https://storm.apache.org/.

Galante, Guilherme and Luis Carlos E de Bona (2012). "A survey on cloud computing elasticity." In: *2012 IEEE fifth international conference on utility and cloud computing*. IEEE, pp. 263–270.

Gandhi, Anshul et al. (2018). "Model-driven optimal resource scaling in cloud." In: *Software & Systems Modeling* 17, pp. 509–526.

Garí, Yisel, David A Monge, and Cristian Mateos (2022). "A Q-learning approach for the autoscaling of scientific workflows in the Cloud." In: *Future Generation Computer Systems* 127, pp. 168–180.

Gedik, Buğra et al. (2013). "Elastic scaling for data stream processing." In: *IEEE Transactions on Parallel and Distributed Systems* 25.6, pp. 1447–1463.

Haslbeck, Jonas MB, Laura F Bringmann, and Lourens J Waldorp (2021). "A tutorial on estimating time-varying vector autoregressive models." In: *Multivariate Behavioral Research* 56.1, pp. 120–149.

Heinze, Thomas et al. (2015). "Online parameter optimization for elastic data stream processing." In: *Proceedings of the sixth ACM symposium on cloud computing*, pp. 276–287.

Herbst, Nikolas Roman, Samuel Kounev, and Ralf H Reussner (2013). "Elasticity in Cloud Computing: What It Is, and What It Is Not." In: *ICAC*. Vol. 13. 2013, pp. 23–27.

Hidalgo, Nicolas, Daniel Wladdimiro, and Erika Rosas (2017). "Self-adaptive processing graph with operator fission for elastic stream processing." In: *Journal of Systems and Software* 127, pp. 205–216.

Hillmer, Steven Craig and George C Tiao (1982). "An ARIMA-model-based approach to seasonal adjustment." In: *Journal of the American Statistical Association* 77.377, pp. 63–70.

Hyndman, Rob J and George Athanasopoulos (2018). *Forecasting: principles and practice*. OTexts.

James, Gareth et al. (2013). *An introduction to statistical learning*. Vol. 112. Springer.

Karim, Md Ebtidaul et al. (2021). "BHyPreC: a novel Bi-LSTM based hybrid recurrent neural network model to predict the CPU workload of cloud virtual machine." In: *IEEE Access* 9, pp. 131476–131495.

Khan, Tahseen et al. (2022). "Machine learning (ML)–Centric resource management in cloud computing: A review and future directions." In: *Journal of Network and Computer Applications*, p. 103405.

Kotze, Kevin (2022). "Vector autoregression models." In.

Kumar, Jitendra and Ashutosh Kumar Singh (2018). "Workload prediction in cloud using artificial neural network and adaptive differential evolution." In: *Future Generation Computer Systems* 81, pp. 41–52.

Liu, Fang et al. (2011). "NIST cloud computing reference architecture." In: *NIST special publication* 500.2011, pp. 1–28.

Liu, Xunyun and Rajkumar Buyya (2017). "D-storm: Dynamic resource-efficient scheduling of stream processing applications." In: *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, pp. 485–492.

Lombardi, Federico et al. (2017). "Elastic symbiotic scaling of operators and resources in stream processing systems." In: *IEEE Transactions on Parallel and Distributed Systems* 29.3, pp. 572–585.

Lorido-Botran, Tania, Jose Miguel-Alonso, and Jose A Lozano (2014). "A review of auto-scaling techniques for elastic applications in cloud environments." In: *Journal of grid computing* 12, pp. 559–592.

Malik, Sania et al. (2022). "A Resource Utilization Prediction Model for Cloud Data Centers Using Evolutionary Algorithms and Machine Learning Techniques." In: *Applied Sciences* 12.4, p. 2160.

Marangozova-Martin, Vania, Noël De Palma, and Ahmed El Rheddane (2019). "Multi-level elasticity for data stream processing." In: *IEEE Transactions on Parallel and Distributed Systems* 30.10, pp. 2326–2337.

Mason, Karl et al. (2018). "Predicting host CPU utilization in the cloud using evolutionary neural networks." In: *Future Generation Computer Systems* 86, pp. 162–173.

Mencagli, Gabriele (2016). "A game-theoretic approach for elastic distributed data stream processing." In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 11.2, pp. 1–34.

Moreno-Vozmediano, Rafael et al. (2019). "Efficient resource provisioning for elastic cloud services based on machine learning techniques." In: *Journal of Cloud Computing* 8.1, pp. 1–18.

Nashaat, Heba, Nesma Ashry, and Rawya Rizk (2019). "Smart elastic scheduling algorithm for virtual machine migration in cloud computing." In: *The Journal of Supercomputing* 75, pp. 3842–3865.

Noshy, Mostafa, Abdelhameed Ibrahim, and Hesham Arafat Ali (2018). "Optimization of live virtual machine migration in cloud computing: A survey and future directions." In: *Journal of Network and Computer Applications* 110, pp. 1–10.

Park, KyoungSoo and Vivek S Pai (2006). "CoMon: a mostly-scalable monitoring system for PlanetLab." In: *ACM SIGOPS Operating Systems Review* 40.1, pp. 65–74.

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Peng, Boyang et al. (2015). "R-storm: Resource-aware scheduling in storm." In: *Proceedings of the 16th annual middleware conference*, pp. 149–161.

Prasad, Vivek Kumar and Madhuri D Bhavsar (2020). "Monitoring and prediction of SLA for IoT based cloud." In: *Scalable Computing: Practice and Experience* 21.3, pp. 349–358.

Rahmanian, Ali Asghar, Mostafa Ghobaei-Arani, and Sajjad Tofighy (2018). "A learning automata-based ensemble resource usage prediction algorithm for cloud computing environment." In: *Future Generation Computer Systems* 79, pp. 54–71.

Reiss, Charles et al. (2012). "Heterogeneity and dynamicity of clouds at scale: Google trace analysis." In: *Proceedings of the third ACM symposium on cloud computing*, pp. 1–13.

Rosa Righi, Rodrigo da et al. (2019). "Elastic-RAN: an adaptable multi-level elasticity model for Cloud Radio Access Networks." In: *Computer Communications* 142, pp. 34–47.

Russo, Gabriele Russo, Valeria Cardellini, and Francesco Lo Presti (2019). "Reinforcement learning based policies for elastic stream processing on heterogeneous resources." In: *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, pp. 31–42.

Russo Russo, Gabriele, Valeria Cardellini, and Francesco Lo Presti (2023). "Hierarchical Auto-Scaling Policies for Data Stream Processing on Heterogeneous Resources." In: *ACM Transactions on Autonomous and Adaptive Systems*.

Russo Russo, Gabriele et al. (2018). "Multi-level elasticity for wide-area data streaming systems: A reinforcement learning approach." In: *Algorithms* 11.9, p. 134.

Shekhar, Shashank et al. (2018). "Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 82–89.

Shen, Siqi, Vincent Van Beek, and Alexandru Iosup (2015). "Statistical characterization of business-critical workloads hosted in cloud datacenters." In: *2015 15th*

IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE, pp. 465–474.

Sherstinsky, Alex (2020). "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network." In: *Physica D: Nonlinear Phenomena* 404, p. 132306.

Shukla, Anshu and Yogesh Simmhan (2018). "Model-driven scheduling for distributed stream processing systems." In: *Journal of Parallel and Distributed Computing* 117, pp. 98–114.

Shyam, Gopal Kirshna and Sunilkumar S Manvi (2016). "Virtual resource prediction in cloud environment: a Bayesian approach." In: *Journal of Network and Computer Applications* 65, pp. 144–154.

Singh, Parminder, Pooja Gupta, and Kiran Jyoti (2019). "TASM: technocrat ARIMA and SVR model for workload prediction of web applications in cloud." In: *Cluster Computing* 22.2, pp. 619–633.

Song, Binbin et al. (2018). "Host load prediction with long short-term memory in cloud computing." In: *The Journal of Supercomputing* 74.12, pp. 6554–6568.

Sun, Dawei et al. (2020). "Dynamic redirection of real-time data streams for elastic stream computing." In: *Future Generation Computer Systems* 112, pp. 193–208.

Thakkar, Riddhi and Madhuri Bhavsar (2022). "Achieving multilevel elasticity for distributed stream processing systems in the cloud environment: A review and conceptual framework." In: *Proceedings of the 2022 Fourteenth International Conference on Contemporary Computing*, pp. 81–90.

— (2023). "NAARPreC: A Novel Approach for Adaptive Resource Prediction in Cloud." In: *Soft Computing and Its Engineering Applications: 4th International Conference, icSoftComp 2022, Changa, Anand, India, December 9–10, 2022, Proceedings*. Springer, pp. 3–16.

Thakkar, Riddhi, Rinni Trivedi, and Madhuri Bhavsar (2017). "Experimenting with energy efficient vm migration in IaaS cloud: Moving towards green cloud." In: *International Conference on Future Internet Technologies and Trends*. Springer, pp. 56–65.

Thakkar, Riddhi Sanjaykumar, Dhyan Thakkar, and Madhuri Bhavsar (2023). "MVMS: RNN based Pro-Active Resource Scaling in Cloud Environment." In: *Scalable Computing: Practice and Experience* 24.1, pp. 17–33.

Thonglek, Kundjanasith et al. (2019). "Improving resource utilization in data centers using an LSTM-based prediction model." In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, pp. 1–8.

Tran, Nhuan et al. (2018). "A multivariate fuzzy time series resource forecast model for clouds using LSTM and data correlation analysis." In: *Procedia Computer Science* 126, pp. 636–645.

Wilkes, John (2011a). "More Google cluster data." In: *Google research blog, Nov.*

— (2011b). *More Google cluster data. Google research blog.* URL: `https://ai.googleblog.com/2011/11/more-google-cluster-data.html`. (accessed: 29 November 2011).

Xu, Le, Boyang Peng, and Indranil Gupta (2016). "Stela: Enabling stream processing systems to scale-in and scale-out on-demand." In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, pp. 22–31.

Xu, Minxian et al. (2022). "EsDNN: Deep Neural Network based Multivariate Workload Prediction Approach in Cloud Environment." In: *arXiv preprint arXiv:2203.02684*.

Yang, Zhengyu et al. (2018). "AutoAdmin: automatic and dynamic resource reservation admission control in Hadoop YARN clusters." In: *Scalable Computing: Practice and Experience* 19.1, pp. 53–68.

Zhu, Yonghua et al. (2019). "A novel approach to workload prediction using attention-based LSTM encoder-decoder network in cloud environment." In: *EURASIP Journal on Wireless Communications and Networking* 2019, pp. 1–18.