Proactive Auto-Scaling Of Containerized Applications In Cloud Environment Using Ensemble-Based Workload Prediction

A Thesis submitted to Gujarat Technological University

for the Award of

Doctor of Philosophy

in

Computer/IT Engineering

by

Trivedi Naimisha Shashikantbhai

Enrollment No.: 179999913019

under supervision of

Dr. Ajaykumar N. Upadhyaya

GUJARAT TECHNOLOGICAL UNIVERSITY

AHMEDABAD

September 2026

© Trivedi Naimisha Shashikantbhai

Declaration

I declare that the thesis entitled **"Proactive Auto-Scaling of Containerized Applications in Cloud Environment Using Ensemble-Based Workload Prediction"** submitted by me for the degree of Doctor of Philosophy is the record of research work carried out by me during the period from **May 2018** to **July 2026** under the supervision and guidance of **Prof. (Dr.) Ajaykumar N. Upadhyaya**, Professor, SAL Engineering & Technical Institute, SAL Education, Ahmedabad and this has not formed the basis for the award of any degree, diploma, associateship, fellowship, titles in this or any other University or other institution of higher learning.

I further declare that the material obtained from other sources has been duly acknowledged in the thesis. I shall be solely responsible for any plagiarism or other irregularities, if noticed in the thesis.

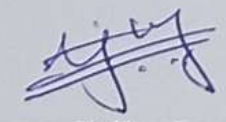
Signature of the Research Scholar: Trivedi Date: 11/09/2026

Name of Research Scholar: **Trivedi Naimisha Shashikantbhai**

Place: Ahmedabad.

CERTIFICATE

I certify that the work in this thesis titled **"Proactive Auto-Scaling of Containerized Applications in Cloud Environment Using Ensemble-Based Workload Prediction"** submitted by **Trivedi Naimisha Shashikantbhai** was carried out by the candidate under my guidance. To the best of my knowledge: (i) the candidate has not submitted the same research work to any other institution for any degree, diploma, Associateship, Fellowship or other similar titles. (ii) the thesis submitted is a record of original research work done by Research Scholar during the period of study under my supervision, and (iii) the thesis represents independent research work on the part of the Research Scholar.

Signature of Research Supervisor:  Date: 11/09/2026

Name of Research Supervisor: **Dr. Ajaykumar N. Upadhyaya**

Place: Ahmedabad.

Course-work Completion Certificate

This is to certify that **Trivedi Naimisha Shashikantbhai** enrolment no. **179999913019** is a PhD scholar enrolled for PhD program in the branch **Computer / IT Engineering** of Gujarat Technological University, Ahmedabad.

(Please tick the relevant option(s))

☐ She has been exempted from the course-work (successfully completed during M.Phil Course)

☐ She has been exempted from Research Methodology Course only (successfully completed during M.Phil Course)

☑ She has successfully completed the PhD course work for the partial requirement for the award of PhD Degree. His performance in the course work is as follows-

Grade Obtained in Research Methodology (PH001)	Grade Obtained in Self Study Course (Core Subject) (PH002)
AB	**AB**

Signature of Research Supervisor:

Name of Research Supervisor: **Dr. Ajaykumar N. Upadhyaya**

Originality Report Certificate

It is certified that PhD Thesis **"Proactive Auto-Scaling of Containerized Applications in Cloud Environment Using Ensemble-Based Workload Prediction"** submitted by **Trivedi Naimisha Shashikantbhai** has been examined by us. We undertake the following:

a. Thesis has significant new work / knowledge as compared already published or are under consideration to be published elsewhere. No sentence, equation, diagram, table, paragraph or section has been copied verbatim from previous work unless it is placed under quotation marks and duly referenced.

b. The work presented is original and own work of the author (i.e. there is no plagiarism). No ideas, processes, results or words of others have been presented as Author own work.

c. There is no fabrication of data or results which have been compiled / analyzed.

d. There is no falsification by manipulating research materials, equipment or processes, or changing or omitting data or results such that the research is not accurately represented in the research record.

e. The thesis has been checked using **Turnitin** (copy of originality report attached) and found within limits as per GTU Plagiarism Policy and instructions issued from time to time (i.e. permitted similarity index $\leq 10\%$).

Signature of the Research Scholar: NTrivedi Date: 11/09/2026

Name of Research Scholar: **Trivedi Naimisha Shashikantbhai**

Place: Ahmedabad.

Signature of Research Supervisor: Ajay Date: 11/09/2026

Name of Research Supervisor: **Dr. Ajaykumar N. Upadhyaya**

Place: Ahmedabad.

Ajay Upadhyaya

PROACTIVE AUTO-SCALING OF CONTAINERIZED APPLICATIONS IN CLOUD ENVIRONMENT USING ENSEMBLE...

Quick Submit
Quick Submit
Gujarat Technological University

Document Details

Submission ID
trn:oid:::1:3609024194

Submission Date
Jul 10, 2026, 12:47 PM GMT+5:30

Download Date
Jul 10, 2026, 12:52 PM GMT+5:30

File Name
NST-Thesis-Without-initialpages-10-7-2026.pdf

File Size
12.4 MB

121 Pages

28,627 Words

170,426 Characters

turnitin Page 1 of 126 - Cover Page Submission ID trn:oid:::1:3609024194

7% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

- Bibliography
- Quoted Text
- Cited Text
- Small Matches (less than 14 words)

Match Groups

- 59 Not Cited or Quoted 7%
  Matches with neither in-text citation nor quotation marks
- 0 Missing Quotations 0%
  Matches that are still very similar to source material
- 0 Missing Citation 0%
  Matches that have quotation marks, but no in-text citation
- 0 Cited and Quoted 0%
  Matches with in-text citation present, but no quotation marks

Top Sources

6% Internet sources
1% Publications
2% Submitted works (Student Papers)

Integrity Flags

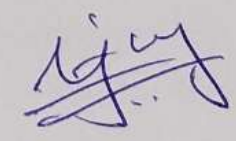
0 Integrity Flags for Review

No suspicious text manipulations found.

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

turnitin Page 2 of 126 - Integrity Overview Submission ID trn:oid:::1:3609024194



PhD THESIS Non-Exclusive License to

GUJARAT TECHNOLOGICAL UNIVERSITY

In consideration of being a PhD Research Scholar at Gujarat Technological University and in the interests of the facilitation of the research work at GTU and elsewhere, I, **Trivedi Naimisha Shashikantbhai** having Enrollment no. **179999913019** hereby grant a non-exclusive, royalty free and perpetual license to GTU on the following terms:

a. The University is permitted to archive, reproduce and distribute my thesis, in whole or in part, and/or my abstract, in whole or in part (referred to collectively as the "Work") anywhere in the world, for non-commercial purposes, in all forms of media;

b. The University is permitted to authorize, sub-lease, sub-contract or procure any of the acts mentioned in paragraph (a);

c. The University is authorized to submit the Work at any National / International Library, under the authority of their "Thesis Non-Exclusive License";

d. The Universal Copyright Notice (©) shall appear on all copies made under the authority of this license;

e. I undertake to submit my thesis, through my University, to any Library and Archives. Any abstract submitted with the thesis will be considered to form part of the thesis.

f. I represent that my thesis is my original work, does not infringe any rights of others, including privacy rights, and that I have the right to make the grant conferred by this non-exclusive license.

g. If third party copyrighted material was included in my thesis for which, under the terms of the Copyright Act, written permission from the copyright owners is required, I have obtained such permission from the copyright owners to do the acts mentioned in paragraph (a) above for the full term of copyright protection.

h. I understand that the responsibility for the matter as mentioned in paragraph (g) rests with the authors/me solely. In no case GTU have any liability for any acts / omissions / errors / copyright infringement from the publication of the said thesis or otherwise.

i. I retain copyright ownership and moral rights in my thesis, and may deal with the copyright in my thesis, in any way consistent with rights granted by me to my University in this non-exclusive license.

j. GTU logo shall not be used/printed in the book (in any manner whatsoever) being published or any promotional or marketing materials or any such similar documents.

k. The following statement shall be included appropriately and displayed prominently in the book or any material being published anywhere: "The content of the published work is part of the thesis submitted in partial fulfillment for the award of the degree of Ph.D. in Computer/IT Engineering of the Gujarat Technological University."

l. I further promise to inform any person to whom I may hereafter assign or license my copyright in my thesis of the rights granted by me to my University in this non-exclusive license. I shall keep GTU indemnified from any and all claims from the Publisher(s) or any third parties at all times resulting or arising from the publishing or use or intended use of the book / such similar document or its contents.

m. I am aware of and agree to accept the conditions and regulations of PhD including all policy matters related to authorship and plagiarism.

Signature of the Research Scholar: NBTrivedi

Name of Research Scholar: **Trivedi Naimisha Shashikantbhai**

Place: Ahmedabad Date: 11/09/2026

Recommendation of the Research Supervisor:

Signature of Research Supervisor:

Name of Research Supervisor: **Dr. Ajaykumar N. Upadhyaya**

Thesis Approval Form

The viva-voce of the PhD Thesis submitted by **Trivedi Naimisha Shashikantbhai** (Enrollment No.:**179999913019**) entitled **"Proactive Auto-Scaling Of Containerized Applications In Cloud Environment Using Ensemble-Based Workload Prediction"** was conducted on 22/09/2026 (Friday),(day and date) at Gujarat Technological University.

(Please tick any one of the following option)

☑ The performance of the candidate was satisfactory. We recommend that he be awarded the PhD degree.

☐ Any further modifications in research work recommended by the panel after 3 months from the date of first viva-voce upon request of the Supervisor or request of Independent Research Scholar after which viva-voce can be re-conducted by the same panel again.

☐ The performance of the candidate was unsatisfactory. We recommend that he should not be awarded the PhD degree.

Dr. Ajaykumar N. Upadhyaya

Name and Signature
of Supervisor with Seal

Dr. Ankit B. Desai

(External Examiner 1)
Name and Signature

Dr. Jayesh Gangrade

(External Examiner 2)
Name and Signature

Abstract

Cloud-native applications increasingly rely on microservices and containerized deployment models to achieve scalability, flexibility, and efficient resource utilization. In such environments, application workloads are highly dynamic and may fluctuate rapidly due to bursty traffic, periodic demand cycles, or continuous growth in user requests. Managing resources elastically is therefore essential to maintain Quality of Service (QoS) and ensure reliable application performance. Traditional reactive auto-scaling mechanisms, which provision additional container replicas only after predefined utilization thresholds are exceeded, are often limited by scaling lag. This delay can lead to transient Service Level Agreement (SLA) violations, increased latency, request rejection, and inefficient utilization of computing resources. This thesis presents a proactive auto-scaling framework for containerized microservices that uses workload prediction to provision container replicas before demand surges occur.

The proposed framework integrates a clustering-enhanced hybrid ensemble workload prediction model with a MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) control loop deployed on Docker Swarm. The hybrid prediction model combines K-means clustering, Prophet, Long Short-Term Memory (LSTM), and Linear Regression, using inverse-error-based weighting to capture seasonal, non-linear, and steady workload patterns. The prediction model is evaluated using RMSE, MAE, MAPE, and R^2 metrics, while auto-scaling performance is assessed in terms of SLA compliance, latency, request rejection, throughput, replica count, and scaling actions. Experiments are conducted using two real workload traces, which include NASA HTTP logs and ClarkNet WWW traces, and two synthetic workload patterns that have monotonically increasing and periodic traffic patterns to evaluate the effectiveness of the proposed framework.

The results show that the proposed proactive auto-scaling framework consistently outperforms reactive auto-scaling across all workload patterns. Compared with the reactive approach, it improves SLA compliance by approximately 5.3 percentage points and reduces mean P95 latency by 6.5–43.3%. Request rejection is also reduced by 68.2%

for the NASA workload, 47.8% for ClarkNet, 9.9% for the increasing workload, and 31.2% for the periodic workload. These improvements are achieved with only a moderate resource overhead of approximately ~3% on average across all workload patterns compared with the reactive approach. The hybrid ensemble prediction model achieves an MAE of 2.492363, an RMSE of 3.415395, and an R^2 of 0.859265 on the NASA dataset and the lowest MAE of 5.806198 and RMSE of 7.3380937 on the ClarkNet dataset among the compared baseline models. These results confirm that proactive auto-scaling reduces scaling delay, improves application performance, and supports efficient resource management in containerized microservice environments.

Keywords: Proactive auto-scaling, container orchestration, Docker Swarm, Workload prediction, Hybrid ensemble model, MAPE-K, SLA compliance.

Acknowledgement

The successful completion of this research is a result of immense support, guidance, and encouragement received from many individuals. Above all, I express my heartfelt gratitude to the Almighty God for granting me the strength, wisdom, and perseverance to successfully complete this research journey. I feel truly privileged to acknowledge and express my sincere thanks to all those who stood beside me and made this journey both meaningful and rewarding.

I express my profound gratitude to my research supervisor, Dr. Ajaykumar N. Upadhyaya, whose mentorship has been invaluable throughout my doctoral journey. His patience, wisdom, and unwavering belief in my abilities not only shaped this research but also nurtured my academic confidence.

I express my sincere gratitude to my former research supervisor, Dr. Shailesh D. Panchal, for his valuable guidance and support during the initial phase of my doctoral journey.

I am eternally grateful to my husband, Mr. Nirav Vyas, my pillar of strength, for his unconditional love, sacrifices, prayers, and constant encouragement. I also lovingly acknowledge my daughter Krishika, whose innocence, patience, and affection brought joy and renewed strength during the most demanding phases of this work. I am deeply grateful to my parents and family, whose values, discipline, and unwavering support laid the strong foundation of my academic journey. Their guidance and emphasis on learning have played a key role in shaping my knowledge and character.

I sincerely thank my Doctoral Progress Committee members, Dr. Ritesh Patel and Dr. Madhuri Bhavsar, for their constructive feedback, insightful suggestions, and continued support, which played a vital role in refining the quality and direction of my research.

Special appreciation is extended to Dr. Jashvant dave, Dr. Nikunj Gamit, Dr. Jignesh Vaniya, Dr. Dipak Patel, Dr. Om Mehta, Dr. Khushali Raval, Dr. M C Zala, Prof. Chetna Chand and all my VGEC collogues for their valuable discussions, innovative ideas, and consistent support. Their guidance has been a source of inspiration during both progress and uncertainty.

I express my sincere gratitude to my Head of Department, Dr. Vibha Patel, for creating a supportive and motivating research environment. I also acknowledge the institute for granting me the opportunity and providing the necessary facilities to pursue my PhD research.

Finally, I extend my sincere gratitude to all those who may not be mentioned individually but whose support, directly or indirectly, contributed to the successful completion of this research.

With heartfelt gratitude,

Trivedi Naimisha Shashikantbhai

Gujarat Technological University

Contents

Abstract xi

Acknowledgement xiii

List of Figures xix

List of Tables xxi

List of Abbreviations xxii

List of Symbols xxvi

1 Introduction 1
- 1.1 Background and Motivation . . . 1
- 1.2 Problem Context . . . 4
- 1.3 Research Problem . . . 5
- 1.4 Research Questions . . . 7
- 1.5 Aim and Objectives of the Research . . . 8
- 1.6 Scope of the Research . . . 8
- 1.7 Research Contributions . . . 9
- 1.8 Organization of the Thesis . . . 10
- 1.9 Summary . . . 11

2 Literature Review 12
- 2.1 Overview . . . 12
- 2.2 Theoretical Background . . . 12
  - 2.2.1 Cloud Computing and Service Models . . . 13
  - 2.2.2 Evolution of Cloud Deployment Models . . . 15
    - 2.2.2.1 Physical Server-Based Deployment . . . 16
    - 2.2.2.2 Virtual Machine-Based Deployment . . . 16
    - 2.2.2.3 Containerized Deployment . . . 17
    - 2.2.2.4 Microservice-Based Cloud-Native Deployment . . . 18
  - 2.2.3 Container Orchestration Platforms . . . 20
  - 2.2.4 CPU-Intensive Application . . . 22
  - 2.2.5 Resource Placement in Auto-Scaling . . . 23
- 2.3 Auto-scaling Fundamentals and Taxonomy . . . 26
  - 2.3.1 Auto-scaling Methods . . . 27
  - 2.3.2 Auto-scaling Mechanism . . . 30
  - 2.3.3 Workload Dynamics and Prediction for Auto-Scaling . . . 31
    - 2.3.3.1 Workload Patterns in Cloud Applications . . . 31
    - 2.3.3.2 Prediction Horizon and Forecasting Granularity . . . 32

2.4 Related Work . . . 34
2.4.1 Reactive auto-scaling Techniques . . . 34
2.4.1.1 Rule-Based and Threshold-Based Approaches . . . 34
2.4.1.2 Control-Theory and Fuzzy-Logic Approaches . . . 35
2.4.1.3 Limitations of Reactive Approaches . . . 35
2.4.2 Proactive auto-scaling Techniques . . . 36
2.4.2.1 Statistical Time-Series Methods . . . 36
2.4.2.2 Classical Machine Learning Approaches . . . 37
2.4.2.3 Deep Learning Approaches . . . 38
2.4.2.4 Transformer-Based Architectures . . . 40
2.4.2.5 Reinforcement Learning Approaches . . . 40
2.4.2.6 Ensemble Methods for Workload Prediction . . . 41
2.4.2.7 Clustering-Enhanced Ensemble Approaches . . . 43
2.4.3 Comparative Summary of Proactive and Hybrid Approaches . . . 44
2.4.4 Research Gap . . . 46
2.5 Summary . . . 47

3 Hybrid Ensemble Workload Prediction Model 48
3.1 Overview . . . 48
3.2 Problem Formulation . . . 49
3.3 Datasets . . . 50
3.3.1 NASA HTTP Logs . . . 50
3.3.2 ClarkNet WWW Logs . . . 51
3.3.3 Quantitative Characteristics of the NASA and ClarkNet Datasets 51
3.4 Data Pre-processing . . . 53
3.5 Feature Extraction . . . 55
3.5.1 Variance . . . 55
3.5.2 Periodicity . . . 55
3.5.3 Rate of Change . . . 55
3.6 Workload Clustering . . . 56
3.7 Workload Classification . . . 58
3.8 Prediction Models . . . 59
3.9 Ensemble Prediction . . . 60
3.10 Complexity computation . . . 64
3.10.1 Time Complexity . . . 65
3.10.2 Space Complexity . . . 65
3.11 Experimental Results . . . 65
3.11.1 Prediction Accuracy: Model Comparison . . . 65
3.11.2 Sensitivity to Window Size and Prediction Horizon . . . 75
3.12 Summary . . . 77

4 Proactive auto-scaling Model 78
4.1 Overview . . . 78
4.2 Resource Estimation in the Plan Phase . . . 79

4.2.1 Capacity Profiling for Estimating T_{max} . . . 80
4.3 Proactive auto-scaling Algorithm . . . 82
4.3.1 Step 1: Prediction Smoothing . . . 83
4.3.2 Step 2: Effective Workload Blending . . . 83
4.3.3 Step 3: Replica Count Estimation with Safety Margin . . . 84
4.3.4 Step 4: Replica Bounds Enforcement . . . 85
4.3.5 Step 5: Cooldown Enforcement . . . 85
4.3.6 Step 6: Downscale Confirmation . . . 85
4.3.7 Step 7: Latency Guard . . . 86
4.3.8 Proactive Algorithm . . . 87
4.4 Reactive auto-scaling Baseline . . . 88
4.5 Comparison of Proactive and Reactive auto-scaling Models . . . 89
4.6 Parameter Sensitivity . . . 90
4.7 Summary . . . 90

5 System Architecture and MAPE-K-Based Experimental Framework 92
5.1 Overview . . . 92
5.2 Proposed System Architecture . . . 93
5.3 MAPE-K-Based Control Loop Realization . . . 95
5.4 Experimental Cluster Infrastructure . . . 98
5.4.1 Experimental Cluster Infrastructure . . . 98
5.4.2 Physical and Virtual Infrastructure . . . 98
5.4.3 Docker Swarm Configuration . . . 99
5.5 Software Stack and Service Deployment . . . 99
5.5.1 Deployed Services . . . 100
5.5.2 Target Application . . . 100
5.5.3 Docker Swarm Service–Task–Container Relationship . . . 101
5.5.4 Extension to Node-Level Scaling . . . 102
5.6 Monitoring and Metrics Collection . . . 103
5.7 Workload Generation . . . 105
5.7.1 Trace-Driven Workload Modeling . . . 106
5.7.2 Dynamic User Control . . . 107
5.8 Experimental Design . . . 109
5.8.1 Experimental Configurations . . . 109
5.8.2 Evaluation Metrics . . . 109
5.9 Auto-scaler Configuration . . . 110
5.10 Experiment Execution Workflow . . . 111
5.11 Summary . . . 114

6 Results and Discussion 115
6.1 Overview . . . 115
6.2 Auto-scaling Performance: Comprehensive Results . . . 115
6.3 SLA Compliance and Latency Analysis . . . 117
6.3.1 NASA HTTP Workload . . . 117

6.3.2 ClarkNet HTTP Workload 117
6.3.3 Increasing Workload 122
6.3.4 Periodic Workload 122
6.4 Resource Efficiency Analysis 122
6.5 Throughput and Request Rejection Analysis 125
6.6 Contribution of Stability Controls 125
6.7 Cross-Workload Synthesis 126
6.8 Summary 127

7 Conclusion and Future Work 128
7.1 Overview 128
7.2 Summary of Research Contributions 128
7.3 Achievement of Research Objectives 129
7.4 Future Research Directions 130
7.5 Concluding Remarks 131

References 133

List of Publications 146

A Appendix A Experimental Dataset Characteristics 147
A.1 NASA HTTP Access Log Dataset 147
A.2 ClarkNet WWW Dataset 147
A.3 Synthetic Workload Traces 148

B Appendix B Experimental Infrastructure Configuration 149
B.1 AWS EC2 Instance Specifications 149
B.2 Docker Stack Deployment Configuration 149
B.3 Flask Application Source Code (Excerpt) 153
B.4 Prometheus Scrape Configuration 154

List of Figures

2.1 Cloud computing service models 15
2.2 Virtual machines vs. containers 17
2.3 Docker swarm service-lifecycle 21
2.4 Resource over-/under-provisioning 26
2.5 Taxonomy for auto-scaling techniques in Cloud environment 27
2.6 Horizontal vs. vertical auto-scaling of containerised services. 29
2.7 Reactive and proactive mechanisms of the auto-scaling technique in a cloud computing environment. 30
2.8 Workload Prediction Representation 33

3.1 Diagram of recursive multi-step forecasting 50
3.2 NASA HTTP requests over time 51
3.3 Clarknet HTTP requests over time 52
3.4 ACF and PACF plots for the NASA and ClarkNet datasets (10-minute intervals). 53
3.5 Website requests distribution per hour for the ClarkNet and NASA datasets. 53
3.6 NASA Cluster 57
3.7 Clarknet cluster 57
3.8 LSTM Architecture 60
3.9 Hybrid Ensemble Approach for Accurate Workload Prediction 63
3.10 Workload Weights Over Time - NASA 63
3.11 Workload Weights Over Time - Clarknet 64
3.12 Actual vs. predicted workload over the complete time series (Hybrid ensemble, $W = 60$ min, $H = 10$ min). 71
3.13 Zoomed comparison of actual vs. predicted workload (Hybrid ensemble, $W = 60$ min, $H = 10$ min). 72
3.14 Representative prediction intervals for NASA and ClarkNet using the hybrid ensemble. 73
3.15 CDF Comparison (Actual vs Hybrid Vs Individual Models for NASA and ClarkNet) 74

5.1 Docker Swarm-Based Auto-scaling Architecture 94
5.2 MAPE-K model for proactive auto-scaling mechanism 97
5.3 Docker Swarm Configuration 99
5.4 Throughput (Requests per Second) measured by Prometheus 104
5.5 95th percentile request latency observed during workload replay 104
5.6 Dynamic replica scaling of the application service 104
5.7 CPU consumption of the prime application service 105
5.8 Dynamic Shape Controller Algorithm 108
5.9 Output Folder structure Showing Performance Test Results 112
5.10 Experiment Logs 113

6.1 NASA HTTP Workload: Latency comparison across Exp A–D. 118
6.2 ClarkNet WWW Workload: Latency comparison across Exp A–D. . . . 119
6.3 Increasing Synthetic Workload: Latency comparison across Exp A–D. . 120
6.4 Periodic Synthetic Workload: Latency comparison across Exp A–D. . . 121
6.5 Container replica count with arrival rate for Reactive (C) and Proactive (D) - NASA HTTP Trace. 123
6.6 Container replica count with arrival rate for Reactive (C) and Proactive (D) - ClarkNet HTTP Trace. 123
6.7 Container replica count with arrival rate for Reactive (C) and Proactive (D) - Monotonically Increasing Workload. 123
6.8 Container replica count with arrival rate for Reactive (C) and Proactive (D) - Periodic Workload. 124

List of Tables

2.1 Comparison of Cloud Deployment Models 19
2.2 Taxonomy of cloud auto-scaling research 28
2.3 Summary of reactive auto-scaling studies 36
2.4 Summary of prediction-driven proactive auto-scaling studies 38
2.5 Summary of hybrid auto-scaling studies 40
2.6 Comparative Summary of Existing Proactive and Hybrid Auto-Scaling Approaches 45

3.1 Parameter values for baseline methods 68
3.2 Parameter values for the proposed hybrid model 69
3.3 Prediction Accuracy on NASA HTTP Logs (W = 60 min, H = 10 min) . 70
3.4 Prediction Accuracy on ClarkNet WWW (W = 60 min, H = 10 min) . . 70
3.5 RMSE Sensitivity to Window Size and Horizon (Hybrid Ensemble Model) 76
3.6 MAE Sensitivity to Window Size and Horizon (Hybrid Ensemble Model) 77

4.1 Comparison of Proactive and Reactive auto-scaling auto-scaling Models 89

5.1 Cluster Node Configuration 98
5.2 Software Stack and Container Images 100
5.3 Monitoring Components Used in the Experimental Framework 103
5.4 Application-Level Metrics Used by the Autoscaler 103
5.5 PromQL Queries Used for Monitoring and Auto-scaling 105
5.6 Proactive Auto-scaler Parameter Configuration (Exp D) 111

6.1 Performance Summary — All Provisioning Strategies and Workload Patterns 116
6.2 Resource Overhead of Proactive (Exp D) Relative to Reactive (Exp C) . 124
6.3 Impact of Stability Controls (Exp C vs Exp D) 126

7.1 Auto-scaling performance: Exp D (proactive) versus Exp C (reactive) across all four workload patterns. 129

A.1 Statistical summary of the NASA HTTP Access Log dataset. 147
A.2 Statistical summary of the ClarkNet WWW dataset. 148

B.1 EC2 instance configuration for all cluster nodes. 149
B.2 Software stack versions used in the experimental platform. 150

List of Abbreviations

ACF Autocorrelation Function

ACO Ant Colony Optimization

API Application Programming Interface

ARIMA Autoregressive Integrated Moving Average

AWS Amazon Web Services

BFD Best Fit Decreasing

Bi-LSTM Bidirectional Long Short-Term Memory

CaaS Container as a Service

cAdvisor Container Advisor

CDF Cumulative Distribution Function

CDT Cooldown Detection Time

CFS Completely Fair Scheduler

CPU Central Processing Unit

CRD Custom Resource Definition

DL Deep Learning

EC2 Elastic Compute Cloud

EMA Exponential Moving Average

FL Federated Learning

GA Genetic Algorithm

GCP Google Cloud Platform

GNN Graph Neural Network

GWO Grey Wolf Optimizer

HBA Honey Badger Algorithm

HHO Harris Hawks Optimization

HPA Horizontal Pod Autoscaler

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

I/O Input/Output

IoT Internet of Things

JSON JavaScript Object Notation

LHHBA Leader Harris Honey Badger Algorithm

LR Linear Regression

LSTM Long Short-Term Memory

MAE Mean Absolute Error

MAPE Mean Absolute Percentage Error

MAPE-K Monitor, Analyse, Plan, Execute over a Knowledge Base

MCDM Multi-Criteria Decision-Making

MILP Mixed Integer Linear Programming

ML Machine Learning

MSE Mean Squared Error

NSGA-II Non-dominated Sorting Genetic Algorithm II

NIST National Institute of Standards and Technology

OS Operating System

PaaS Platform as a Service

PM Physical Machine

PPO Proximal Policy Optimization

PSO Particle Swarm Optimization

QoE Quality of Experience

QoS Quality of Service

RAM Random Access Memory

RL Reinforcement Learning

RMSE Root Mean Squared Error

RPS Requests Per Second

RT Response Time

SaaS Software as a Service

SLA Service Level Agreement

SLO Service Level Objective

SVR Support Vector Regression

SVM Support Vector Machine

TCN Temporal Convolutional Network

TFT Temporal Fusion Transformer

URL Uniform Resource Locator

VARIMA Vector Autoregressive Integrated Moving Average

VM Virtual Machine

VNF Virtual Network Function

VPA Vertical Pod Autoscaler

WOA Whale Optimization Algorithm

WSGI Web Server Gateway Interface

List of Symbols

Symbol	Description
$A(t)$	Observed request arrival rate (RPS) at time t
$\hat{A}(t+H)$	Predicted workload H minutes into the future
W	Historical observation window (minutes)
H	Prediction horizon (minutes)
f_o	Trained ensemble forecasting model
Var_t	Variance feature of workload window
Per_t	Periodicity feature of workload window
Roc_t	Rate-of-change feature of workload window
$\mathbf{f}_t$	Workload feature vector $[\text{Var}_t, \text{Per}_t, \text{Roc}_t]^\top$
K	Number of K-means clusters (typically 3)
J	K-means objective function
S_i	Set of workload windows assigned to cluster i
c_i	Centroid of cluster i
$d_i(t)$	Distance from feature vector to cluster centroid i
$w_i(t)$	Cluster-proximity weight for model i
$w_{t,k}^{\text{error}}$	Inverse-error weight for base model k at time t
$w_{t,k}$	Final ensemble weight for base model k
$P_{\text{Prophet}}(t)$	Prophet model prediction at time t
$P_{\text{LSTM}}(t)$	LSTM model prediction at time t
$P_{\text{LR}}(t)$	Linear Regression model prediction at time t
$A_s(t)$	Smoothed prediction (EMA output)
λ	Prediction smoothing factor (EMA weight)
$A_{\text{eff}}(t)$	Effective workload used for replica estimation

Symbol	Description
β	Blending factor (predicted vs. observed contribution)
$R^*(t)$	Estimated required replica count
$T_{\max}$	Per-replica throughput capacity (RPS)
α	Safety margin factor
$R(t)$	Current active replica count
$R_{\min}$	Minimum replica count
$R_{\max}$	Maximum replica count
T_c	Cooldown period (seconds)
t_{last}	Timestamp of the last scaling action
$v_a(t)$	Downscale confirmation counter
k_a	Required downscale confirmations
$L_{95}(t)$	95th-percentile response latency at epoch t
L_{SLA}	SLA latency threshold
MAE	Mean Absolute Error
RMSE	Root Mean Squared Error
MAPE	Mean Absolute Percentage Error
R^2	Coefficient of Determination
F	Composite fitness function (LHHBA)
E_{load}	Load prediction error
C_{res}	Resource cost
U_{cpu}	CPU utilisation
w_1, w_2, w_3	Weights for fitness components

Chapter 1

Introduction

1.1 Background and Motivation

Modern digital services depend on the ability of computing infrastructure to process user requests within acceptable time limits. Many applications deployed today are not limited to simple web-page delivery; they also execute CPU-intensive operations such as data compression, encryption, multimedia processing, scientific computation, analytics, recommendation generation, and machine-learning inference. These tasks may consume significant processor resources, especially when multiple users or jobs compete for the same execution capacity. When the available processing capacity becomes insufficient, applications may experience higher response latency, reduced throughput, request queuing, and degraded user experience. Therefore, efficient management of computing resources is essential for maintaining responsiveness under variable and computation-heavy workloads.

The practical importance of maintaining low application latency extends beyond technical performance concerns and directly affects user engagement and business outcomes. Empirical evidence from large-scale web performance studies consistently demonstrates that even sub-second improvements in page responsiveness produce measurable commercial impact. Deloitte reported that a 0.1-second improvement in mobile site speed was associated with an 8.4% increase in conversion rates for retail websites and a 10.1% increase for travel websites, while also reducing average order values dropped due to slow loading [1]. In a controlled A/B experiment, Vodafone observed that improving Largest Contentful Paint by 31% resulted in an 8% increase in sales, a 15% improvement in lead-to-visit rate, and an 11% improvement in cart-to-visit rate [2]. Google has further documented that 53% of mobile users abandon a page that takes longer than three seconds

to load, and that a one-second delay in mobile page load time can reduce conversions by up to 20% [3]. These observations indicate that application responsiveness is not only a technical performance concern, but also an important factor in service effectiveness.

Cloud computing provides an important foundation for addressing this challenge. The National Institute of Standards and Technology (NIST) defines cloud computing as a model that enables convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort [4]. This model allows users to acquire computing, storage, and networking resources when required and release them when demand decreases. As a result, cloud computing reduces dependence on fixed infrastructure and supports dynamic resource provisioning for modern web, enterprise, data-intensive, and cloud-native applications.

A central objective of cloud resource management is elasticity, which refers to the ability of a system to adjust allocated resources according to changing workload and performance requirements [5]. In practice, elasticity is commonly achieved through auto-scaling mechanisms that automatically add, remove, or reconfigure resources such as virtual machines, containers, pods, or service replicas [6, 7]. If resources are provisioned for peak demand, a large portion of the infrastructure may remain idle during low-load periods, resulting in unnecessary cost. If resources are provisioned only for average demand, the application may become overloaded during high-demand periods, causing performance degradation and SLA violations. Hence, auto-scaling must balance application performance, resource utilization, and operational cost.

Designing an effective auto-scaling mechanism is challenging because scaling decisions involve several interdependent factors: when to trigger scaling, which resources to scale, how much capacity to add or remove, and how to execute the action without creating instability [6]. Many auto-scaling mechanisms rely on infrastructure-level metrics such as CPU utilization and memory utilization because they are easy to monitor and directly reflect resource pressure. However, infrastructure-level metrics alone may not fully capture application-level QoS. Therefore, response time, request rate, and SLA-related

latency should also be considered so that scaling decisions reflect both resource usage and user-perceived performance [8].

The deployment model of cloud applications has also changed significantly. Earlier cloud deployments commonly used virtual machines as the primary unit of resource provisioning. Virtual machines provide strong isolation, but they are relatively heavy because each VM usually includes a complete guest operating system. Containerization provides a lighter deployment model by packaging application code and dependencies while sharing the host operating system kernel. Docker popularized this approach by enabling lightweight and portable application containers [9]. Compared with traditional VM-based deployment, containers can reduce deployment overhead and support faster scaling of application components [10].

The adoption of containers and microservices has made resource management more fine-grained. In a microservice architecture, an application is decomposed into smaller services that can be deployed and scaled independently. This improves flexibility, but it also increases the complexity of placement and scaling decisions because different services may have different workload patterns, resource requirements, and performance constraints. Container orchestration platforms such as Kubernetes and Docker Swarm automate deployment, scheduling, service discovery, health monitoring, and scaling, but the quality of resource management still depends on the decision logic used for placement and scaling actions [11].

The motivation of this research arises from these challenges. Efficient cloud auto-scaling requires more than simply adding resources after overload occurs. Resources must be placed and allocated efficiently to reduce cost and wastage; workload trends must be analysed to support proactive decisions; and scaling actions must be planned and executed through a stable closed-loop control process. This thesis addresses these needs by combining workload prediction, capacity-based scaling decision logic, and MAPE-K-based execution for containerized cloud applications.

1.2 Problem Context

The problem addressed in this thesis is situated at the intersection of workload-aware scaling, and closed-loop execution for containerized cloud applications. Modern cloud platforms provide mechanisms for deploying and scaling application instances, but the effectiveness of these mechanisms depends on the decision logic used to allocate resources, anticipate demand, and execute scaling actions. Therefore, the thesis does not treat auto-scaling as a single isolated task. Instead, it considers auto-scaling as a sequence of connected decisions: where resources should be placed, how future demand should be estimated, how many replicas should be planned, and how the final action should be executed.

One part of the problem concerns workload dynamics and the challenge of accurate prediction. Cloud workloads may exhibit different temporal behaviors, including periodic demand, bursty arrivals, gradual growth, and relatively steady low-variance intervals. These patterns are difficult to model using a single forecasting method because seasonal, non-linear, and stable workload segments require different modelling capabilities. Therefore, an effective proactive auto-scaling approach must analyse workload variation and estimate future demand in a manner that is robust across heterogeneous workload patterns.

Another part of the problem concerns implementation in a containerized environment. Application-level metrics such as request rate and response latency must be considered together with infrastructure-level metrics such as CPU and memory utilization, because resource pressure alone may not fully represent user-perceived QoS [8]. In this thesis, these concerns are organized using a MAPE-K-inspired closed-loop structure, where monitoring, analysis, planning, execution, and knowledge work together to support adaptive resource management [7, 12].

Formal Problem Formulation: Let $A(t) \in \mathbb{R}^+$ denote the observed request arrival rate (RPS) at time t, and let $\hat{A}(t+H)$ denote the predicted workload H minutes into the future. The workload predictor is defined as a forecasting function that uses the recent historical workload window $\hat{A}(t+H) = f_\theta(A(t-W+1), A(t-W+2), \ldots, A(t))$ where W

is the historical observation window and f_θ is the trained forecasting model. Given the predicted workload, the number of required replicas is: $R^*(t) = \left\lceil \frac{\hat{A}(t+H)}{T_{\max}.\alpha} \right\rceil$ where $T_{\max}$ is the per-replica throughput capacity (RPS) and $\alpha \in (0, 1)$ is a safety margin factor. The term $T_{\max}.\alpha$ represents the effective usable capacity of one replica. This prevents the autoscaler from assuming that each replica can continuously operate at its absolute maximum throughput.

The autoscaling objective is to determine a replica assignment policy $\pi : \mathcal{S} \to \mathcal{A}$ that minimises the expected SLA violation rate and reduce response latency while keeping the replica count within the available cluster capacity:

$$\min \left[\mathbb{E}\left(\frac{1}{T} \sum_{t=1}^{T} L_{95}(t) \right) + \eta \mathbb{E}\left(\frac{1}{T} \sum_{t=1}^{T} \mathbf{1}[L_{95}(t) > L_{\text{SLA}}] \right) \right]$$

$$\text{subject to} \quad R_{\min} \leq R^*(t) \leq R_{\max}, \quad \forall t$$

where $L_{95}(t)$ is the 95th-percentile response latency at epoch t, L_{SLA} is the SLA threshold, $R(t)$ is the average replica count, and $R_{\max}$ is the cluster capacity, S is the system state and A is the scaling action space.

The core research challenge is to construct an accurate workload predictor $\hat{A}(t+H) = f_\theta(A(t-W+1), \ldots, A(t))$ that minimises prediction error across heterogeneous workload patterns, and enables the autoscaler to provision replicas ahead of demand, thereby reducing scaling delay, lowers p95 response latency, minimizes SLA violations, and improves resource efficiency.

1.3 Research Problem

Containerized cloud applications operate under dynamic workload conditions in which request demand, resource utilization, and application-level performance may vary over time. Although container orchestration platforms provide mechanisms for deployment, scheduling, monitoring, and scaling, the effectiveness of these mechanisms depends on the decision models used to analyse workload demand and determine scaling actions.

Reactive auto-scaling mechanisms remain widely used because of their simplicity, but they may initiate scaling only after resource pressure or application-level degradation has already occurred. This scaling delay may lead to temporary under-provisioning, increased p95 latency, request rejection, and SLA violations.

Proactive auto-scaling attempts to reduce this delay by estimating future workload demand and provisioning container replicas before severe overload occurs. However, prediction alone is not sufficient for effective auto-scaling. The predicted workload must be translated into a feasible target replica count using per-replica capacity, safety margin, replica limits, cooldown control, and scale-in protection. Without these mechanisms, an auto-scaler may over-provision resources, under-provision capacity, or produce unstable scaling behaviour.

Another challenge concerns the interpretation of runtime metrics. Infrastructure-level metrics such as CPU utilization and memory utilization indicate resource pressure, but they do not always capture application-level quality of service. Application-level indicators such as request rate, response time, response latency, rejected requests, and SLA-related performance provide additional information about user-perceived service quality. Therefore, an effective auto-scaling framework must relate workload demand, resource availability, application-level performance, and scaling stability within a coordinated decision process.

The research problem addressed in this thesis is formulated as follows.

> How can a containerized cloud application be managed through a proactive auto-scaling framework that combines workload prediction, capacity-based replica planning, stability-aware control, and closed-loop execution in order to reduce scaling delay, improve application-level performance, and maintain efficient resource usage?

This research problem is decomposed into different technical sub-problems. First, a workload prediction model is required to estimate future request demand across heterogeneous workload patterns. Second, a scaling decision model is required to

convert predicted demand into a feasible target replica count under system constraints and stability controls. Third, the scaling action must be executed through a closed-loop framework capable of monitoring application and container metrics, analyzing runtime state, planning adaptation, applying scaling actions, and maintaining relevant system knowledge.

Accordingly, this thesis investigates an integrated approach in which a clustering-enhanced hybrid ensemble model supports workload prediction, a capacity-based planning model computes target replica count, and a MAPE-K-inspired control loop organizes runtime execution on Docker Swarm.

1.4 Research Questions

Based on the research problem, this thesis is guided by the following research questions:

1. **RQ1:** How can heterogeneous cloud workload patterns, including seasonal, bursty, increasing, and steady traffic, be represented and predicted using a clustering-enhanced hybrid ensemble model?

2. **RQ2:** How can predicted workload demand be converted into a stable target number of container replicas using capacity-based planning, safety margin, replica bounds, cooldown control, and scale-in protection?

3. **RQ3:** To what extent does the proposed proactive auto-scaling framework reduce response latency, request rejection, and SLA violations compared with no scaling, static provisioning, and reactive auto-scaling strategies?

4. **RQ4:** How can the proposed proactive auto-scaling framework be implemented and evaluated in a Docker Swarm-based containerized environment using Prometheus, cAdvisor, and trace-driven workload generation?

1.5 Aim and Objectives of the Research

The aim of this research is to design, implement, and evaluate an integrated resource management framework for containerized cloud applications by combining optimization-based ensemble-based workload prediction, and MAPE-K-based proactive auto-scaling. The research focuses on improving resource efficiency and application-level performance by supporting cost-aware resource allocation and provisioning container replicas ahead of workload demand.

To achieve this aim, the following objectives are defined:

1. To design a clustering-enhanced hybrid ensemble model for accurate workload prediction in cloud environments by combining K-means clustering with Prophet, LSTM, and linear regression using inverse-error-based dynamic weighting.
2. To develop a MAPE-K-based proactive auto-scaling framework that uses multi-step-ahead workload forecasts to provision container replicas before demand materializes, thereby reducing reactive scaling lag.
3. To implement and deploy a real containerized testbed on Docker Swarm with Prometheus-based monitoring, cAdvisor container metrics, and trace-driven workload generation using Locust, enabling reproducible end-to-end evaluation.
4. To evaluate and compare four provisioning strategies, namely no scaling, static provisioning, reactive auto-scaling, and proactive auto-scaling, across real and synthetic workload patterns in order to assess application-level performance, SLA compliance, request-handling capability, resource usage, and scaling stability.

1.6 Scope of the Research

The scope of this research is defined as follows.

Inclusions

- Containerized microservices are deployed on Docker Swarm for the proactive auto-scaling framework, with horizontal scaling.
- A CPU-bound stateless microservice is used as the target application.
- Real-world HTTP workload traces, namely NASA and ClarkNet, along with two synthetic workload patterns, are used for workload generation and evaluation.
- Application-level monitoring, including request rate and p95 latency, and container-level monitoring, including CPU utilization and replica count, are considered.
- Offline model training is performed with online MAPE-K control-loop execution.
- Reactive auto-scaling is used as the primary baseline for comparison with the proposed proactive auto-scaling approach.

Exclusions

- Vertical scaling, including CPU and memory resource-limit adjustment per container, and node scaling are not considered in real testbed.
- Multi-tier microservice architectures with inter-service dependencies.
- Online model retraining or concept drift adaptation during execution.
- Memory-bound or I/O-bound workload types.
- Kubernetes or other orchestration platforms.
- Cost-based optimization using public cloud pricing models.

1.7 Research Contributions

The primary contributions of this research are as follows:

1. **Hybrid Ensemble Prediction Model:** A workload prediction model is developed by combining K-means clustering, Prophet, LSTM, and linear regression with inverse-error-based dynamic weighting.

2. **Proactive Auto-Scaling Framework for Docker Swarm:** A MAPE-K-based proactive auto-scaling framework is developed for Docker Swarm, integrating workload prediction, latency awareness, and stability controls.

3. **Reproducible Experimental Testbed:** A three-node AWS EC2 Docker Swarm testbed is implemented using Prometheus, cAdvisor, and trace-driven Locust workload generation.

4. **Multi-Workload Comparative Evaluation:** A systematic evaluation is conducted across NASA HTTP, ClarkNet HTTP, increasing, and periodic workloads using no scaling, static provisioning, reactive auto-scaling, and proactive auto-scaling strategies.

1.8 Organization of the Thesis

The thesis is organized as follows.

Chapter 2 presents the background concepts, literature review, taxonomy of auto-scaling approaches, and research gap identification.

Chapter 3 describes the hybrid ensemble workload prediction model and evaluates its forecasting accuracy with different window size and horizons.

Chapter 4 presents the proactive auto-scaling decision model that converts predicted workload into a target container replica count.

Chapter 5 describes the Docker Swarm-based MAPE-K implementation, including monitoring, workload generation, and scaling execution.

Chapter 6 presents the experimental results and comparative analysis across no scaling, static provisioning, reactive auto-scaling, and proactive auto-scaling strategies.

Chapter 7 concludes the thesis by summarizing the findings, contributions, limitations, and future research directions.

1.9 Summary

This chapter introduced the background, motivation, and problem context of resource management and proactive auto-scaling for containerized cloud applications. It presented the research problem, aim and objectives, scope, major contributions, and organization of the thesis. The chapter established the need for an integrated approach that combines workload prediction, capacity-based scaling decisions, and MAPE-K-based closed-loop execution. The next chapter presents the domain background, literature review, taxonomy of existing auto-scaling approaches, and research gap identification.

Chapter 2

Literature Review

2.1 Overview

This chapter presents a comprehensive review of the literature on auto-scaling of cloud applications, with particular emphasis on proactive approaches that use workload prediction to anticipate changes in resource demand before performance degradation occurs.

The review is organized into six major themes. It begins by introducing the fundamentals of cloud computing, including its service models and the evolution of deployment models from physical servers to virtual machines, containers, and microservices. It then discusses the concepts and taxonomy of auto-scaling techniques, followed by a review of reactive auto-scaling approaches and their limitations. The chapter subsequently examines proactive auto-scaling techniques based on statistical time-series models, classical machine learning, deep learning, transformer-based architectures, reinforcement learning, and ensemble methods. Finally, it reviews resource placement and optimization in elastic scaling, along with the MAPE-K autonomic control loop and its application in containerized cloud environments.

2.2 Theoretical Background

This section introduces the fundamental concepts of cloud computing, its service models, and the evolution of deployment paradigms relevant to auto-scaling strategies.

2.2.1 Cloud Computing and Service Models

Cloud computing is a distributed computing paradigm that enables users to access configurable computing resources over a network on an on-demand basis. According to the National Institute of Standards and Technology (NIST), cloud computing provides convenient network access to a shared pool of configurable resources, such as networks, servers, storage, applications, and services, that can be rapidly provisioned and released with minimal management effort or service-provider interaction [4]. The cloud model is characterized by five essential properties like on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service [4]. These characteristics make cloud computing suitable for applications with dynamic workload requirements because resources can be acquired, scaled, and released according to demand.

Cloud computing has significantly changed the way computing infrastructure is provisioned and consumed. Instead of purchasing and maintaining dedicated physical infrastructure, users can obtain virtualized computing resources from cloud providers and pay according to usage. This model improves resource availability, reduces upfront infrastructure investment, and supports elastic scaling of applications [13, 14]. From the perspective of resource management, elasticity is one of the most important features of cloud computing because it allows applications to adapt their resource capacity when workload demand changes. However, elasticity does not occur automatically without appropriate resource management mechanisms. It requires monitoring, decision-making, provisioning, and execution mechanisms that can adjust resources efficiently while maintaining application performance.

Cloud services are commonly classified into different service models according to the level of abstraction and management responsibility provided to the user. The three classical cloud service models are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [4]. In addition, the increasing adoption of containerized deployment has led to the emergence of Container as a Service (CaaS), where container orchestration and management are offered as a cloud service.

In *Infrastructure as a Service* (IaaS), the cloud provider offers virtualized computing, storage, and networking resources. Users can provision virtual machines, configure operating systems, install middleware, and deploy applications according to their requirements. IaaS provides the highest level of control among the major cloud service models, but it also places greater management responsibility on the user. Virtual machines are typically the main provisioning unit in IaaS environments, and resource management is commonly performed by adding, removing, or resizing VM instances.

In *Platform as a Service* (PaaS), the cloud provider offers a managed platform that includes the operating environment, runtime, middleware, and development tools required for application deployment. Users mainly manage application code and data, while the provider manages most infrastructure-level concerns. PaaS reduces operational complexity compared with IaaS and allows developers to focus on application logic. Many PaaS systems also provide built-in support for deployment automation and application scaling.

In *Software as a Service* (SaaS), the complete application is delivered to end users over the network. Users access the application through a web browser or client interface and do not manage the underlying infrastructure, platform, runtime, or application code. SaaS represents the highest level of abstraction from the user perspective. Examples include web-based productivity tools, enterprise applications, customer relationship management systems, and online collaboration platforms.

With the growth of containerized and microservice-based deployment, *Container as a Service* (CaaS) has become an important service model. CaaS provides facilities for deploying, managing, networking, and scaling containers across a cluster. Unlike IaaS, where virtual machines are the primary unit of provisioning, CaaS focuses on containers, tasks, pods, or service replicas as the operational unit. Container orchestration platforms such as Docker Swarm and Kubernetes support this model by maintaining desired service state, scheduling containers on available nodes, providing service discovery, and enabling horizontal scaling of application replicas [9, 11].

The distinction among cloud service models is important for auto-scaling research because each model exposes different scaling units and control responsibilities. In IaaS, auto-scaling is commonly performed at the virtual-machine level. In PaaS and CaaS

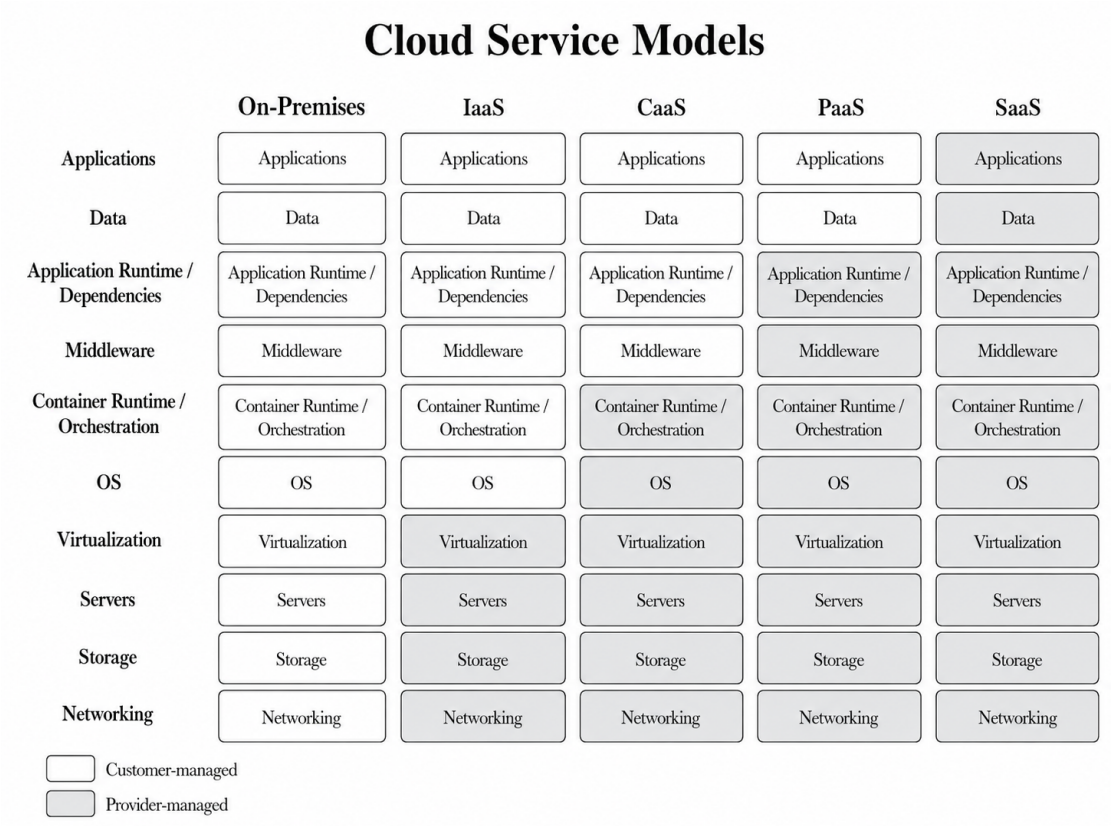


FIGURE 2.1: Cloud computing service models

environments, auto-scaling can be applied at the application, container, pod, or service-replica level. In SaaS, scaling is generally hidden from end users and handled internally by the provider. Therefore, the service model determines what can be monitored, what can be scaled, and how scaling decisions are executed. Since this thesis focuses on containerized applications, the main scaling unit considered in the later chapters is the container service replica.

2.2.2 Evolution of Cloud Deployment Models

The deployment of cloud applications has evolved from fixed physical infrastructure to virtualized, containerized, and microservice-oriented environments. This evolution is important for auto-scaling research because the deployment model determines the unit of scaling, the provisioning delay, the level of isolation, and the degree of resource management flexibility.

2.2.2.1 Physical Server-Based Deployment

In physical server-based deployment, an application is installed directly on dedicated bare-metal hardware. The operating system, application runtime, libraries, and application code share the same physical machine. This model was common before widespread virtualization and cloud computing. It provides direct access to hardware resources, but it lacks flexibility. Scaling requires purchasing, installing, and configuring new hardware, which may take significant time and administrative effort.

Another limitation of physical deployment is poor resource elasticity. Infrastructure is usually provisioned for expected peak demand. During low-demand periods, a large portion of the hardware may remain idle, resulting in poor utilization. During unexpected demand peaks, additional capacity may not be available quickly, causing degraded performance. Therefore, physical server-based deployment is not suitable for dynamic workload environments where demand changes frequently.

2.2.2.2 Virtual Machine-Based Deployment

Virtualization introduced a major improvement over physical server deployment by allowing multiple virtual machines to run on the same physical host [15]. A hypervisor abstracts the physical hardware and provides isolated virtual environments. Each VM contains its own guest operating system, application runtime, and application code as shown in Figure 2.2. This model improves resource sharing, isolation, and manageability compared with dedicated physical servers.

VM-based deployment became the foundation of early cloud computing platforms. In this model, users can provision or terminate VMs according to application demand. auto-scaling is commonly performed by adding new VMs, removing unused VMs, or changing the resource capacity of existing VMs. Horizontal VM scaling increases the number of VM instances, while vertical VM scaling increases or decreases resources such as CPU and memory allocated to existing VMs.

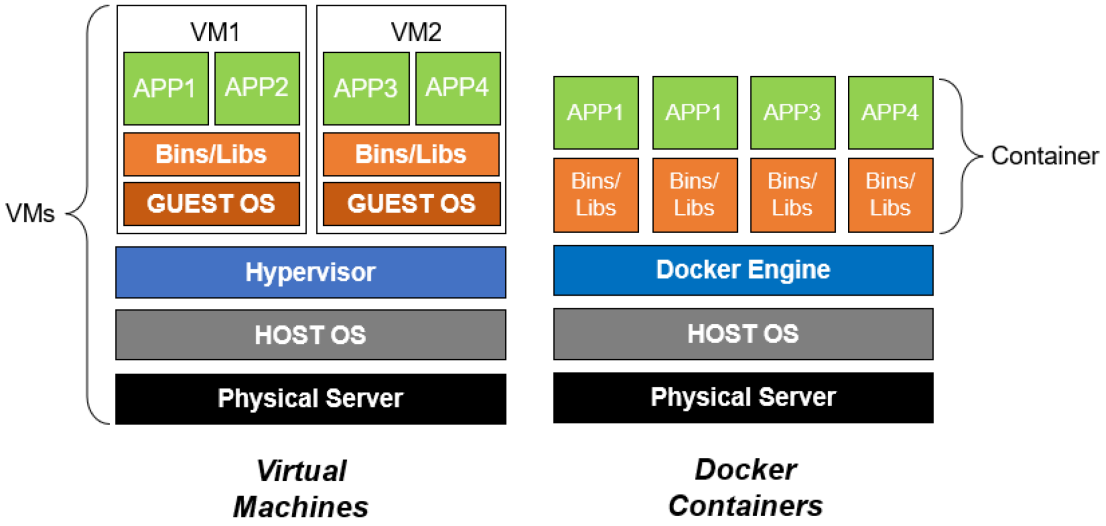


FIGURE 2.2: Virtual machines vs. containers [18]

VM-based scaling suffers from significant provisioning latency: booting a new VM typically requires 60–300 seconds depending on image size and cloud provider [10]. This startup penalty makes reactive VM scaling particularly slow to respond to rapid demand changes, motivating research into prediction-assisted approaches [16, 17].

2.2.2.3 Containerized Deployment

Containerized deployment provides a lighter alternative to virtual machines [9]. A container packages application code, runtime libraries, and dependencies, but shares the host operating system kernel. As a result, containers are smaller and faster to start than VMs. They also provide portability because the same container image can be executed across different environments that support the container runtime.

In containerized deployment, multiple containers can run on the same host with lower overhead than multiple VMs. This makes containers suitable for applications that require rapid deployment, frequent updates, and fine-grained scaling. Instead of scaling an entire VM, a system can scale individual service containers or replicas. This improves resource efficiency and allows faster response to workload changes. Google container engine, Amazon Elastic Container Service and Microsoft Azure Container Service are examples of widely used platforms.

Morabito et al. [10] experimentally demonstrated that containers achieve 2–5× lower startup latency and 10 to 20% higher throughput density compared to KVM-based VMs. Singh et al. [19] quantified container elasticity, introducing a Cooldown Detection Time (CDT) mechanism to mitigate scaling oscillation while maintaining fast elastic response.

The lighter footprint of containers makes horizontal scaling the dominant mode for stateless microservices. However, even with sub-minute container startup, the combined detection and provisioning delay remains non-trivial for latency-sensitive applications [20]. Thus, containerized scaling still requires careful control because frequent replica changes can cause instability, scheduling overhead, and oscillation. Therefore, the planning model must include stability mechanisms such as cooldown, replica bounds, downscale confirmation, and latency guard.

2.2.2.4 Microservice-Based Cloud-Native Deployment

Modern cloud computing environments increasingly rely on cloud-native applications, which are purposefully designed to leverage the elasticity, scalability, and resilience of the cloud [21]. While traditional software relies on monolithic architectures where components are tightly coupled within a single system, the cloud-native paradigm favors microservice-based deployment. This approach decomposes an application into a collection of smaller, autonomous services, each responsible for a distinct business or functional capability. Because each microservice is developed, deployed, and scaled independently, communication is offloaded to lightweight protocols such as RESTful APIs, Remote Procedure Calls (RPC), or asynchronous message queues. This architectural style provides the foundational infrastructure for cloud-native modularity, localized fault isolation, and independent software development lifecycles.

A primary advantage of microservices within cloud-native applications is granular scalability. Instead of scaling a monolithic application as a single, resource-heavy unit, individual components, such as payment gateways, search engines, or user profile services, can scale dynamically according to their unique workload patterns. This isolated scaling prevents over-provisioning, minimizes operational costs, and optimizes underlying infrastructure resource allocation.

TABLE 2.1: Comparison of Cloud Deployment Models

Parameter	**Physical / Bare Metal**	**Virtual Machines (VMs)**	**Containers**	**Serverless (FaaS)**	**Microservice-Based Cloud-Native**
Isolation Level	Hardware-level isolation	Hypervisor-level isolation	Process-level (shared kernel) isolation	Provider-managed isolation	Service-level isolation (via containers)
Startup Time	Minutes to hours	30 seconds to several minutes	Milliseconds to a few seconds	Milliseconds (may have cold-start delay)	Milliseconds to a few seconds
Resource Overhead	Full hardware dedicated	High (full guest operating system)	Low (shares host operating system kernel)	Minimal (charged only for actual execution)	Low (lightweight independent services)
Density	One application per server	Multiple VMs per physical server	Dozens to hundreds of containers per host	Thousands of concurrent function instances	High (many small independent services)
Portability	Dependent on hardware	Dependent on hypervisor and VM image	High (standard container images)	Limited (tied to cloud provider)	High (service-independent deployment)
Scalability	Manual provisioning	Moderate (slower scale-out)	Fast horizontal scaling	Near-instant automatic scaling	Fine-grained independent scaling per service
Operational Overhead	Hardware + OS management	OS patching and VM lifecycle management	Container orchestration and monitoring	Almost none (fully managed by provider)	Medium (service discovery, orchestration, monitoring)
Typical Use Case	Legacy systems, specialized hardware	Traditional enterprise applications	Microservices and cloud-native applications	Event-driven and bursty workloads	Large-scale distributed cloud-native systems

Despite these advantages, microservice-based cloud-native deployment makes auto-scaling more complex [21]. A microservice application usually consists of several services that communicate with one another. Therefore, scaling only one service may not always improve the overall application performance, because another dependent service may become the next bottleneck. In addition, infrastructure-level metrics such as CPU utilization may not fully represent the actual performance experienced by users. For this reason, application-level metrics such as request rate, response latency, error rate, queue length, and SLA compliance are also important for auto-scaling decisions.

To keep the experimental environment controlled and reproducible, this thesis focuses on a stateless CPU-intensive microservice. This design avoids the additional complexity

of multi-service dependency analysis and allows the study to focus on horizontal replica scaling. Since the workload is CPU-bound, an increase in request rate directly increases CPU pressure and response latency, making it suitable for evaluating the effect of proactive auto-scaling on latency reduction and SLA preservation.

Since the proposed framework is implemented in a containerized environment, the next subsection discusses container orchestration platforms, which provide the runtime mechanisms for scheduling, monitoring, and scaling containerized services.

2.2.3 Container Orchestration Platforms

As containerized applications grow in size and complexity, manual container management becomes impractical. Container orchestration platforms address this issue by automating container deployment, scheduling, networking, service discovery, health monitoring, and scaling. An orchestration platform maintains the desired state of a service. If a service is defined with a certain number of replicas, the orchestrator attempts to keep that number of running containers available across the cluster. Several orchestration platforms have been used to manage containerized applications, including Kubernetes, Docker Swarm, Apache Mesos with Marathon, OpenShift, and HashiCorp Nomad.

Kubernetes [11] and Docker Swarm [22] are two famous container orchestration platforms. Kubernetes provides a rich ecosystem and advanced resource management features, including pod scheduling, service discovery, rolling updates, and auto-scaling mechanisms. Docker Swarm is integrated with Docker Engine and provides a simpler service model based on managers, workers, overlay networks, services, tasks, and replicas. Its relatively simple configuration and direct service scaling interface make it suitable for controlled experimental studies. In this thesis, Docker Swarm is selected as the orchestration platform because it supports service replication and scaling in a lightweight cluster environment. [23] has systematically compared Docker Swarm with Kubernetes, OpenShift, and Mesos as container orchestration platforms for cloud-based applications.

In Docker Swarm, an application is deployed as a service, and the number of desired replicas defines how many task-container instances should be running. When the replica

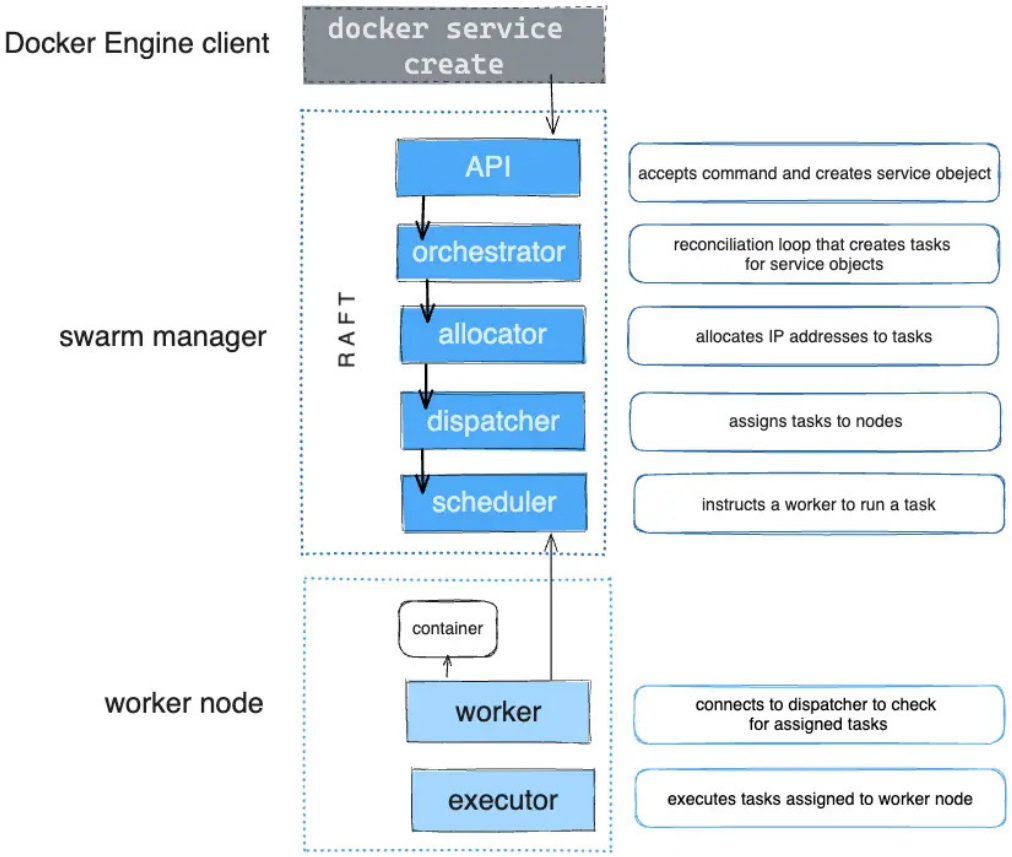


FIGURE 2.3: Docker swarm service-lifecycle [22]

count is updated, the Swarm manager schedules new tasks or removes existing tasks to match the desired state. This service-replica model is directly relevant to the proposed auto-scaling framework because the output of the Plan phase is a target replica count. The Figure 2.3 shows how Swarm mode accepts service create requests and schedules tasks to worker nodes. Chapter 5 describes the Docker Swarm implementation details, while this section establishes the conceptual background needed to understand why container orchestration is appropriate for proactive auto-scaling.

Apache Mesos provides a general cluster-resource management layer, with Marathon commonly used for long-running containerized services. OpenShift builds on Kubernetes with additional enterprise platform services, while Nomad provides a lightweight scheduler for containers and non-containerized workloads.

In addition to the deployment and orchestration model, the type of application workload also affects auto-scaling behaviour. The following subsection discusses the relevance of

CPU-bound workloads in auto-scaling evaluation.

2.2.4 CPU-Intensive Application

This study uses a CPU-intensive containerized application as the target service for auto-scaling evaluation. CPU-intensive applications, also knows as Compute-intensive applications, are widely used in cloud computing and their execution behavior is strongly related to processor demand. Peng et al.[24] modeled CPU-intensive applications in cloud environments and showed that such applications have identifiable characteristics that distinguish them from other application types. In a related study, Peng et al. [25] further argued that scheduling and processing strategies should consider the type of application, because CPU-intensive applications require specific handling to improve execution efficiency .

A CPU-intensive workload is suitable for this research because its performance is directly affected by the availability of CPU resources. As the incoming request rate increases, each request consumes processor time, CPU utilization rises, request queues build up, and response latency increases. Therefore, insufficient replica capacity can be clearly observed through CPU saturation and response latency degradation.

[26] studied auto-scaling for CPU-intensive containerized applications and showed that auto-scaling decisions can be driven by performance measures collected from different layers of the container execution stack. Their work investigates how to select suitable performance measures for triggering auto-scaling actions while satisfying QoS constraints, and analyzes how CPU usage measures influence resource allocation decisions under different workload conditions.

Motivated by these studies, the present work uses a stateless CPU-bound containerized service as the experimental application. Since increasing RPS directly increases CPU pressure and response latency, this workload provides a controlled and meaningful environment for evaluating the impact of proactive auto-scaling on latency reduction and SLA preservation.

2.2.5 Resource Placement in Auto-Scaling

In cloud computing, auto-scaling involves dynamically adjusting resources to match varying workload demands. However, deciding the number of resources (scaling) is only one part of the problem. The equally critical challenge is where to place these new resources (VMs or containers) within a heterogeneous cloud environment consisting of physical machines (PMs) with varying CPU, memory, storage, and network capacities. In such heterogeneous settings, inefficient placement can lead to severe resource fragmentation, load imbalance, higher operational costs, and poor overall system utilization—even when the correct number of resources has been provisioned through scaling.

This creates a complex joint optimization problem: how to simultaneously determine the appropriate scaling action (horizontal or vertical) and find the optimal placement of resources to minimize cost while satisfying performance and capacity constraints. Traditional rule-based or threshold-based auto-scaling approaches often ignore this placement aspect, delegating it to default schedulers that do not consider cost or long-term efficiency. Metaheuristic optimization techniques have emerged as promising solutions to address this integrated scaling and placement challenge in heterogeneous environments. Classical heuristics such as First Fit, Best Fit, Round Robin, and Kubernetes scheduling strategies are commonly used as baseline methods, while GA, PSO, ACO, WOA, HHO, and HBA represent the major metaheuristic families used for placement and consolidation problems.

The prevailing approaches to cloud scaling face several significant challenges. Bottleneck detection mechanisms proposed in [27] enable scaling decisions based on resource congestion; however, they lack integration of horizontal and vertical scaling strategies, limiting their applicability for large-scale resource allocation. Although integrated scaling approaches using optimization techniques such as integer programming were introduced in [28], they rely primarily on CPU and memory metrics and do not incorporate comprehensive workload prediction, thereby affecting accuracy. This limitation was partially addressed by the CPN-based elastic controller in [29], which improved resource estimation under varying workloads, but it did not fully exploit auto-scaling mechanisms

for performance enhancement. Similarly, the StreamScale-H framework proposed in [30] demonstrated efficient elastic scaling; however, it lacked validation on real cloud platforms and integration with open-source stream processing systems.

Overall, existing scaling techniques struggle to adapt to highly dynamic and unpredictable workloads, where fixed models become ineffective. Furthermore, achieving flexible resource allocation on physical machines while maintaining Quality of Service (QoS) remains a critical challenge in cloud environments [31].

Building upon this body of work, the preliminary phase of the present research led to the development of the Leader Harris Honey Badger Algorithm (LHHBA), a hybrid metaheuristic optimization technique that integrates the leader-based exploration and mutation strategy of Leader Harris Hawks Optimization (LHHO) with the foraging behaviour of the Honey Badger Algorithm (HBA). The LHHBA algorithm was evaluated in a simulated VM-based cloud environment using Bitbrains VM resource traces. It employed VARIMA for multivariate load prediction on CPU utilization, memory utilization, disk, and network metrics.

Based on the predicted resource demand, a two-threshold decision mechanism was adopted to determine the appropriate scaling strategy. When the predicted demand remained below the vertical scaling threshold, no scaling action was performed. If the demand exceeded the vertical threshold but remained below the horizontal threshold, vertical scaling was initiated through resource expansion of the existing virtual machine. Conversely, when the predicted demand surpassed the horizontal threshold, horizontal scaling was triggered by provisioning additional virtual machines.

LHHBA then optimized the resource allocation using exploration and exploitation phases controlled by the escape energy parameter. Separate solution representations and cost-based fitness functions were used for horizontal and vertical scaling. The fitness functions considered CPU, memory, disk, and software license costs while ensuring that the capacity constraints of the physical machines were satisfied. The pseudocode is presented in Algorithm 1.

Algorithm 1 Leader Harris Honey Badger Algorithm (LHHBA)

Require: Population size, maximum iterations, scaling thresholds

Ensure: Optimal resource allocation

```
Initialize hawk population
Evaluate fitness of each solution
Identify the best solution
while termination criterion is not satisfied do
    Update escape energy
    for each hawk do
        if exploration phase then
            Update position using adaptive perch probability and honey-mode search
        else
            Perform surprise pounce using soft or hard besiege strategy
            Apply leader-based mutation and selection
        end if
        Evaluate the updated solution
        Update the best solution
    end for
end while
return Optimal scaling allocation
```

Simulation results showed competitive performance with low prediction error compared to moving average and linear regression models and reduced scaling cost compared to several baselines research papers like [27, 28, 29, 32].

However, the simulator-based study had key limitations. It focused on VM-to-PM allocation in a simulated environment and did not address VM-to-container placement, which is central to this thesis. The evaluation considered only increasing workloads, excluding realistic mixed or real-time patterns, and lacked validation on real container orchestration platforms.

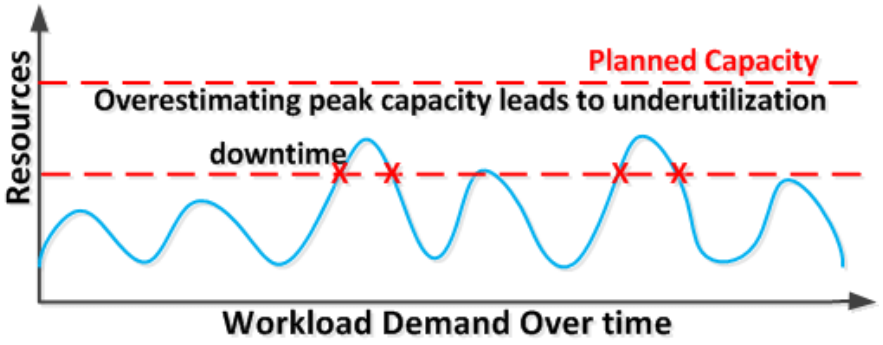


FIGURE 2.4: Resource over-/under-provisioning [35]

These gaps motivated the present work, which advances toward proactive auto-scaling using a clustering-enhanced hybrid ensemble prediction model and validates it on a real Docker Swarm–based testbed.

The concepts presented in this section form the basis for understanding the challenges and opportunities in auto-scaling. The next section examines auto-scaling fundamentals, taxonomy, and a detailed review of existing reactive and proactive techniques.

2.3 Auto-scaling Fundamentals and Taxonomy

Auto-scaling refers to the automatic adjustment of computing resources in response to changes in application workload. In cloud environments, this is a practical expression of elasticity: resources are added when demand rises and released when demand falls [5, 33, 34]. This capability is necessary because cloud workloads are rarely constant. They may change with user activity, peak-hour access, seasonal behavior, background events, failures, or sudden traffic bursts. A system configured for peak load may remain idle for long periods, wasting resources and increasing cost. In contrast, a system configured for average load may not have enough capacity when demand rises unexpectedly as shown in Figure 2.4.

This imbalance between demand and allocated capacity is one of the main reasons for SLA violations. When incoming requests exceed the available resources, CPU utilisation rises, queues begin to build up, and response time increases. As a result, the application may fail to satisfy SLA requirements such as maximum response time,

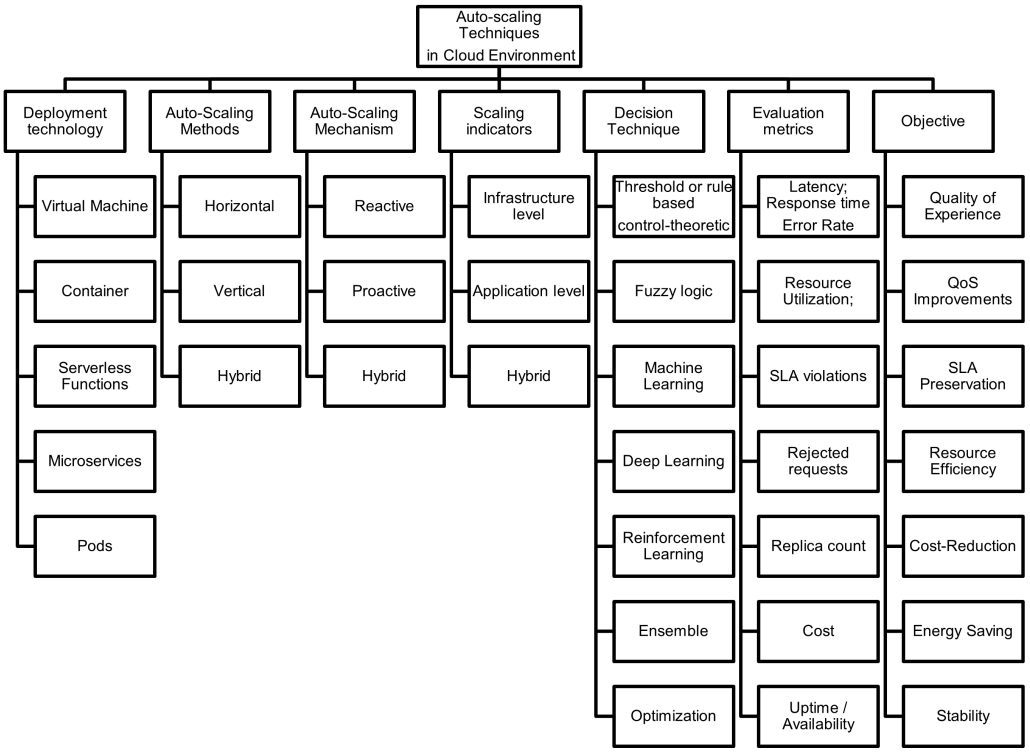


FIGURE 2.5: Taxonomy for auto-scaling techniques in Cloud environment

minimum availability, acceptable latency, or permitted violation rate. Such violations can affect user experience, increase operational penalties, and reduce trust in the service provider. auto-scaling helps address this problem by adjusting resources continuously, so that performance remains within SLA limits without keeping excessive capacity active during normal workload conditions.

Cloud auto-scaling research can be classified from several perspectives. A complete classification needs to consider the deployment technology, scaling direction, monitored indicators, decision-making method, evaluation metrics, and intended objectives. Therefore, this thesis adopts a seven-dimensional taxonomy for analysing existing auto-scaling studies as shown in Figure 2.5. Table 2.2 summarizes the main dimensions used to classify cloud auto-scaling research.

2.3.1 Auto-scaling Methods

Auto-scaling operates along three orthogonal dimensions (Figure 2.6):

TABLE 2.2: Taxonomy of cloud auto-scaling research

Category	**Purpose**
Deployment technology	Defines the execution environment and scaling unit (e.g., VMs; containers; microservices; serverless functions; POD).
Auto-Scaling Methods	Defines whether scaling changes the number of instances, the capacity of existing instances, or both (horizontal; vertical; hybrid).
Auto-Scaling Mechanism	Defines whether scaling occurs after metric changes, before predicted demand, or through a combined strategy (reactive; proactive; hybrid).
Scaling indicators	Defines the metrics used to trigger or guide scaling decisions (CPU/memory utilization; request rate; throughput; response time; tail latency; SLA/SLO violations).
Decision technique/ Approach	Defines how the required scaling action is computed (threshold/rule-based; control-theoretic; fuzzy logic; forecasting; ML; DL; RL; ensemble; optimization).
Evaluation metrics	Defines how auto-scaling effectiveness is measured (latency/response time; error rate; resource utilization; SLA violations; rejected requests; replica count; cost; oscillation; uptime/availability).
Objective	Defines the intended benefit of the auto-scaling approach (QoS improvement; SLA preservation; resource efficiency; cost reduction; stability).

- **Horizontal scaling (scale-out/in):** Horizontal scaling, or scale-out/scale-in, changes the number of execution units used to serve the application workload. In VM-based systems, this means adding or removing VM instances, usually behind a load balancer. This approach provides strong isolation but may involve longer provisioning and startup delays because each VM includes a guest operating system. In container-based systems, horizontal scaling changes the number of containers, pods, tasks, or service replicas.

  Since containers share the host operating system kernel, they are generally lighter and faster to start than VMs. Kubernetes performs this form of scaling by adjusting pod replicas, while Docker Swarm adjusts the desired replica count of a service. Thus, Docker Swarm service-replica scaling is a controlled form of horizontal container scaling.

- **Vertical scaling (scale-up/down):** Vertical scaling, also referred to as scale-up or scale-down, adjusts the resource capacity of an existing execution unit instead of changing the number of running instances. In VM-based cloud environments,

vertical scaling typically involves modifying the vCPU, memory, or storage capacity allocated to a virtual machine. This can be achieved through cold resizing, where the VM is stopped and restarted with a different instance configuration, or through live scaling mechanisms such as CPU and memory hot-plugging when supported by the hypervisor and guest operating system. In container-based environments, vertical scaling modifies resource limits or reservations of containers or pods, such as CPU shares, CPU quota, memory limits, or Kubernetes resource requests and limits. Kubernetes VPA, Al-Dhuraibi et al. [35, 36] and Dawoud et al. [37] are examples of vertical auto-scaling mechanisms in container-based environments. Both approaches are bounded by host capacity and may require restart or platform-specific support when resources such as memory are changed.

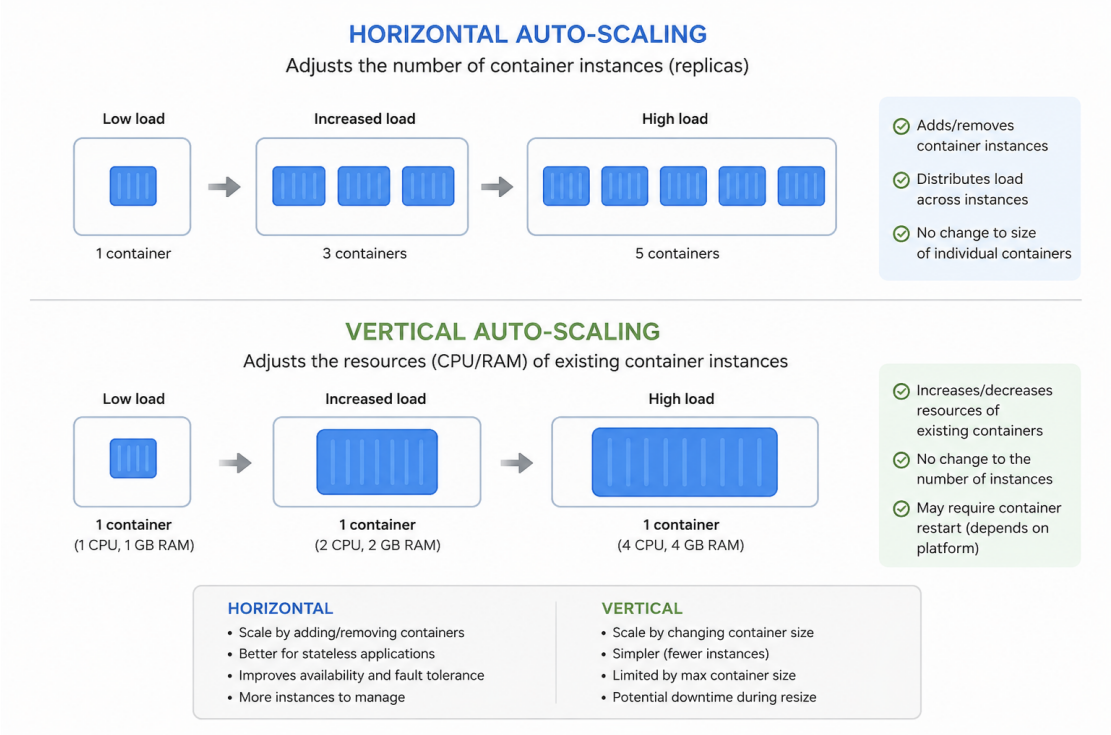


FIGURE 2.6: Horizontal vs. vertical auto-scaling of containerised services.

- **Hybrid scaling:** Hybrid scaling combines horizontal and vertical scaling within the same resource-management policy. In general, vertical scaling is used for fine-grained adjustment of the resource capacity of an existing VM, container, or service instance, whereas horizontal scaling is used for coarse-grained capacity expansion by adding or removing execution units. Several studies have explored

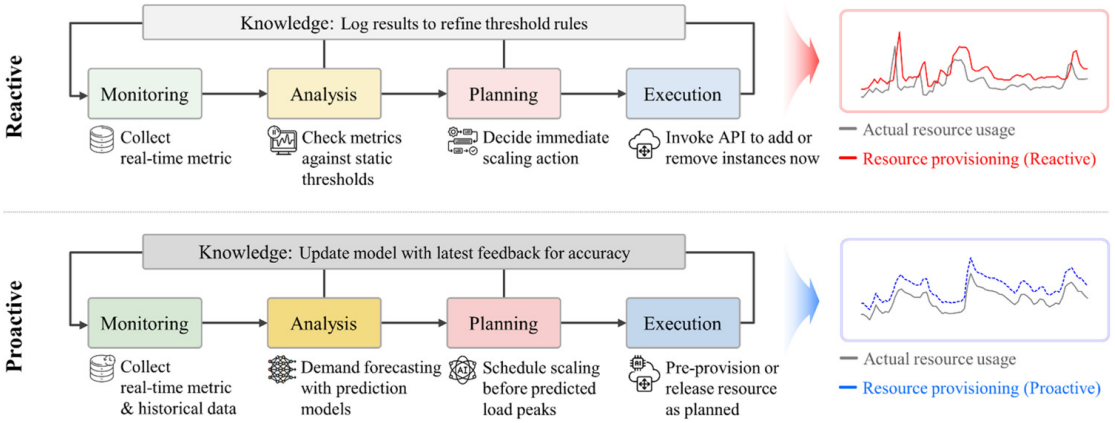


FIGURE 2.7: Reactive and proactive mechanisms of the auto-scaling technique in a cloud computing environment. [42]

hybrid scaling strategies [38, 39, 40, 41] because neither horizontal nor vertical scaling alone is sufficient for all workload conditions.

2.3.2 Auto-scaling Mechanism

In cloud computing, the mechanism that determines the scaling timing of auto-scaling techniques is generally composed of reactive, proactive, and hybrid mechanisms, as shown in Figure 2.7.

1. **Reactive:** Reactive auto-scaling mechanisms rely on continuous real-time monitoring of system metrics such as resource utilization, queue length, and HTTP request rates to make scaling decisions. These algorithms are simple to develop and deploy; however, the time taken to scale resources leads to degradation of resource availability and violates SLA compliance [43].

2. **Proactive (predictive):** Proactive mechanism-based auto-scaling performs scaling based on predicted values to overcome the limitations of reactive-based auto-scaling through a process known as "cold starting" [44]. This approach removes the latency inherent in scaling resources. Therefore, proactive-based auto-scaling must predict future workloads or future scaling decisions in order to proactively accommodate real-time workloads.

3. **Hybrid:** Combines a proactive predictor for anticipated demand with a reactive fallback for unpredicted bursts. Although proactive auto-scaling reduces scaling delay by forecasting future demand, it is not sufficient by itself in production environments. Real workloads may include bursts, anomalies, and irregular changes that cannot always be predicted accurately. Straesser et al. [45] argue that production-ready auto-scaler should combine proactive and reactive components, because prediction can prepare resources in advance, while reactive control can respond to unexpected SLO violations or prediction errors. They also highlight that a key challenge in such hybrid systems is the aggregation of proactive and reactive outputs into a single scaling decision.

   The proposed framework adopts a hybrid proactive–reactive strategy by combining predicted and observed request rates using a weighting factor. Prediction enables early scaling decisions, while real-time observations correct inaccuracies and handle sudden workload changes. This ensures both responsiveness and robustness in dynamic cloud environments.

2.3.3 Workload Dynamics and Prediction for Auto-Scaling

Workload prediction is an important component of proactive auto-scaling because scaling actions require time to take effect. If future demand can be estimated before overload occurs, resources can be provisioned in advance, and the impact of scaling delay can be reduced. However, cloud workloads are dynamic and may change according to user behaviour, time-of-day effects, seasonal events, background tasks, or sudden traffic bursts. Therefore, prediction for auto-scaling must consider both the temporal behaviour of workload and the practical time required to execute a scaling action.

2.3.3.1 Workload Patterns in Cloud Applications

Cloud application workloads may exhibit different temporal patterns [46]. A steady workload remains relatively stable over time, while a periodic workload shows repeated

daily, weekly, or seasonal behaviour. A bursty workload contains sudden increases in demand, and an increasing workload shows gradual growth over time.

A steady workload typically occurs in long-running backend services such as database servers or enterprise systems, where user requests remain relatively constant. Periodic workloads arise in applications with predictable usage cycles, such as e-commerce platforms experiencing daily traffic peaks during evening hours, or enterprise systems showing higher activity on weekdays compared to weekends.

Bursty workloads are commonly observed during real-world events such as flash sales, viral social media trends, breaking news, or ticket booking openings, where a sudden surge of user requests leads to rapid spikes in demand. Increasing workloads, on the other hand, occur in scenarios such as growing user bases, system adoption phases, or long-running data processing pipelines where the workload gradually intensifies over time.

These patterns are important because different prediction models may perform differently across workload types. For example, models that capture seasonality may be suitable for periodic traces, whereas non-linear models may be more suitable for irregular or bursty traces. This motivates the use of pattern-aware and ensemble-based prediction methods in proactive auto-scaling.

2.3.3.2 Prediction Horizon and Forecasting Granularity

The prediction horizon defines how far into the future the workload is estimated. In auto-scaling, the horizon should be long enough to cover monitoring delay, prediction time, scheduling delay, container startup time, and application warm-up time. If the horizon is too short, the system may still react late; if it is too long, prediction uncertainty may increase [47]. Forecasting granularity defines the time interval at which workload is aggregated and predicted, such as seconds, minutes, or longer intervals. Fine-grained prediction can support faster adaptation, but it may be more sensitive to noise. Coarser prediction is more stable but may hide short bursts.

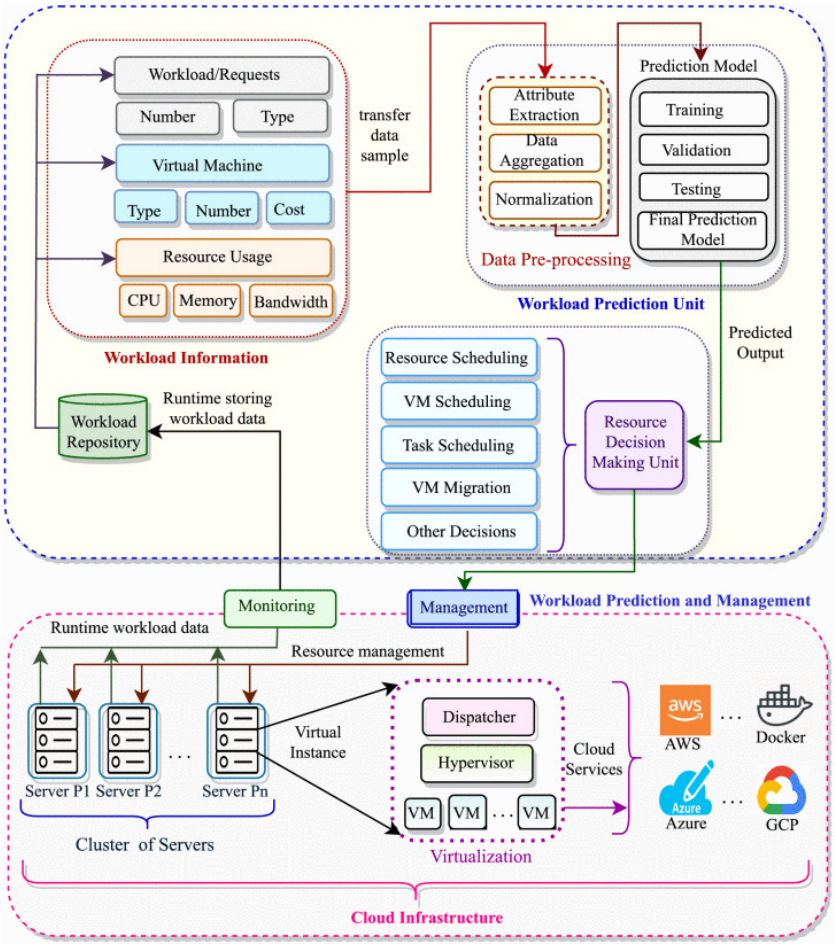


FIGURE 2.8: Workload Prediction Representation [48]

Workload prediction plays a crucial role in enabling proactive resource management in cloud environments. As illustrated in Figure 2.8, workload-related information, including request characteristics, virtual machine configurations, and resource usage metrics (CPU, memory, and bandwidth), is continuously monitored and stored in a workload repository. This data is pre-processed through attribute extraction, aggregation, and normalization before being fed into a prediction model. The model, trained and validated using historical data, forecasts future workload demands, which are then utilized by the resource decision-making unit for efficient scheduling, migration, and allocation of resources. This predictive approach helps in minimizing provisioning delays and ensures improved Quality of Service (QoS) [48].

Despite its advantages, workload prediction faces several challenges. These include handling heterogeneous workloads with varying resource requirements, managing uncertain

and bursty demand patterns, adapting to dynamic cloud environments, and determining appropriate data granularity and prediction horizons. Addressing these challenges is essential for improving the accuracy and effectiveness of predictive auto-scaling mechanisms.

2.4 Related Work

2.4.1 Reactive auto-scaling Techniques

Reactive auto-scaling represents the production baseline. The Kubernetes Horizontal Pod Autoscaler (HPA) monitors CPU utilisation at 15-second intervals and adjusts pod count when utilisation deviates from a target (default 50%) [49].

2.4.1.1 Rule-Based and Threshold-Based Approaches

et al. [20] conducted the comprehensive analysis of HPA operational behaviour, comparing Pod Count Mode (PCM) and Kubernetes Resource Metric (KRM). They found that PCM scales to 24 replicas while KRM reaches only 21 for the same workload, and that a Readiness Probe eliminates failed requests by delaying traffic routing until containers are ready.

Singh et al. [19] quantified the elasticity of container-based auto-scaling using a CDT interval that suppresses premature scale-in. Their work showed that improperly configured CDT leads to either resource waste (too long) or oscillation (too short). Taherizadeh et al. [50] investigated key influencing factors of HPA for computing-intensive applications, demonstrating that the scraping interval, stabilisation window, and CPU target percentage all significantly affect scaling behaviour. Node-level auto-scaling solutions like reactive Kubernetes node autoscaler (RKNA) dynamically adjust cluster size by analyzing node-level metrics and selecting optimal scaling targets using multi-criteria decision-making (MCDM) [51].

Srirama et al. [52] proposed container auto-scaling with warm-container reuse to reduce cold-start latency, though their approach lacked oscillation prevention and remained vulnerable to burst workloads. The cost-efficient elastic process scaling framework of [53] formulated auto-scaling as a Mixed Integer Linear Programming (MILP) optimisation, achieving simultaneous cost reduction and SLA compliance.

2.4.1.2 Control-Theory and Fuzzy-Logic Approaches

Xu et al. [54] developed a fuzzy-logic autoscaler for fog computing that achieved lower latency and error rates than rule-based scaling, though it required manual tuning of fuzzy membership functions. The HiDRA framework [55] introduced hierarchical decentralised reactive auto-scaling for spatio-temporal burst traffic in fog environments, using a multi-level control hierarchy to distribute scaling decisions. Arabnejad et al. [56] combined fuzzy logic with Q-learning in OpenStack, enabling the autoscaler to adapt without threshold configuration.

The Self-Adaptive Requirements-Driven framework [57] proposed specialised scaling strategies for different SLO conditions, providing efficient operation through condition-specific policies. While it improved upon basic HPA, the reactive nature still incurs unavoidable detection delay.

2.4.1.3 Limitations of Reactive Approaches

The fundamental limitation shared by all reactive approaches is the *scaling lag/cold start problem*—the time between the onset of demand increase and the availability of new capacity. This lag comprises: (i) metric scrape interval (15–60 s), (ii) stabilisation window (30–300 s), (iii) container scheduling and startup (10–120 s), and (iv) application initialisation [58]. During this combined delay, the service operates under-provisioned, causing elevated latency and potential SLA violations. Reactive auto-scaling also cannot distinguish between transient spikes and sustained demand increases, leading to either premature scaling or delayed response.

TABLE 2.3: Summary of reactive auto-scaling studies

Ref.	Approach	Scaling Indicators	Scaling Type	Evaluation Metrics	Key Limitation
[57]	Control-based	Resource utilization, SLO violation	Hybrid	Resource Utilization, number of instances, response time, SLO violation rate	Requires further optimization for different SLO adjustments.
[19]	Rule-based	Resource utilization	Horizontal	Resource Utilization, scaling speed	Inappropriate cooldown detection time may cause resource waste or degradation.
[51]	Rule-based	Resource utilization, energy	Horizontal	Resource Utilization, processing time	Node-level scaling increases operational complexity.
[50]	Rule-based	Resource utilization	Horizontal	Resource Utilization, response time, instances	Limited adaptation to nonlinear workload patterns.
[52]	Rule-based, heuristic	Utilization, queue length	Horizontal	Processing time, cost, utilization, instances	Vulnerable to burst workloads and lacks oscillation prevention.
[53]	Control-based	Utilization, SLA violation	Hybrid	Utilization, cost, SLA violation	MILP-based optimization may face scalability issues.
[54]	Fuzzy control	Resource utilization	Horizontal	Delay, error rate, cost	Requires tuning of fuzzy parameters.
[55]	Control-based	Utilization, QoS violation, request distribution	Horizontal	Success rate, instances, network overhead	Initial learning may make optimal scaling difficult.

2.4.2 Proactive auto-scaling Techniques

Proactive auto-scaling uses resource or workload prediction to anticipate future demand and initiate scaling actions before the demand surge materialises. This section reviews prediction-based approaches organised by the underlying technique: statistical methods, classical machine learning, deep learning, and transformer-based architectures.

2.4.2.1 Statistical Time-Series Methods

Classical time-series methods such as ARIMA, exponential smoothing, and their variants were among the earliest approaches applied to workload prediction for auto-scaling.

Box and Jenkins [59] established the ARIMA methodology that remains a benchmark in workload forecasting. Shariffdeen et al. [60] developed the inteliScaler system for PaaS clouds, combining ARIMA with exponential smoothing and regression models in an ensemble, supplemented by cost modelling and "smart kill" strategies. Their follow-up work demonstrated that ensemble forecasting yields significantly lower errors than any single model and outperforms the Apache Stratos built-in predictor [60].

Roy et al. [16] proposed an efficient auto-scaling algorithm combining look-ahead optimisation with workload prediction that reduced SLA violations while minimising VM-hours. Calheiros et al. [61] demonstrated ARIMA-based workload prediction for cloud resource provisioning, achieving meaningful reduction in SLA violations over reactive baselines.

While statistical methods offer low computational cost and interpretability, they struggle with non-linear patterns, sudden regime changes, and long-range dependencies that characterise modern cloud workloads [17].

2.4.2.2 Classical Machine Learning Approaches

Machine learning (ML) approaches such as Support Vector Regression (SVR), Random Forests (RF), and Decision Trees offer non-linear modelling capability beyond ARIMA.

Iqbal et al. [62] proposed dynamic workload pattern prediction using K-means clustering to partition URI traffic into patterns, then applying Non-Negative Least Squares (NNLS) regression for each pattern. Their Pattern-Window Prediction (PWP) approach reduced prediction error across all four identified traffic types.

Al Qassem et al. [63] developed a proactive Random Forest autoscaler that achieves 90% improvement in resource utilisation and 95% reduction in p95 latency for microservices. Their RF ensemble regression on Bitbrains fastStorage traces demonstrated RMSE reduction compared to single-model baselines. Yadav et al. [48] used SVR within a MAPE-K loop for proactive horizontal container scaling, sustaining performance under fluctuating workloads.

Abdullah et al. [64] applied Decision Tree Regression for predictive auto-scaling in fog microdata centres, achieving 77.53% reduction in SLO violations and 75.51% reduction in rejected requests on Wikipedia and FIFA traces compared to reactive approaches.

TABLE 2.4: Summary of prediction-driven proactive auto-scaling studies

Ref.	Technique	Scaling Indicators	Scaling Type	Evaluation Metrics	Key Limitation
[63]	ML	Resource utilization	Hybrid	Utilization, latency, response time	Prediction accuracy may degrade for unseen workload patterns.
[65]	DL	Resource utilization	Horizontal	Resource Utilization, latency	May oscillate under highly volatile workloads.
[66]	DL	Resource utilization	Horizontal	Resource Utilization, cost, SLA violation	Sudden workload changes remain difficult to predict.
[67]	DL, GNN	Resource utilization	Horizontal	SLA violation, cost	Requires modelling of service dependencies and may be complex.
[68]	DL	Resource utilization	Horizontal	Over-provisioning, under-provisioning	Multiple models increase computational cost.
[69]	DL	Utilization, bandwidth	Hybrid	Latency	Prediction accuracy may remain limited despite correction.
[70]	DL, FL	Utilization, traffic load	Hybrid	Instances, round-trip time	Federated learning may reduce accuracy compared with centralized models.
[71]	DL, RL, fuzzy logic	Resource utilization	Horizontal	Processing time, utilization, energy	Training and operation require additional computational cost.
[72]	RL	Utilization, response time	Hybrid	Response time, cost, utilization	Limited suitability for memory-bound applications.
[73]	RL	Invocation rate	Horizontal	Response time, utilization, latency	Training requires additional computational resources.
[74]	RL	Utilization, latency, throughput	Hybrid	Utilization, SLO violation	Adaptation may not guarantee optimality in all scenarios.

2.4.2.3 Deep Learning Approaches

Deep learning models, particularly LSTM, Bi-LSTM, and Temporal Convolutional Networks (TCN), have demonstrated superior performance in capturing temporal dependencies.

1. **LSTM and Bi-LSTM.**

Hochreiter and Schmidhuber [75] introduced LSTM networks that address the vanishing gradient problem in RNNs, enabling learning of long-range temporal dependencies. Imdoukh et al. [76] pioneered LSTM-based proactive auto-scaling for containerised applications, demonstrating that LSTM matches ARIMA prediction accuracy with approximately 600× faster inference speed.

Dang-Quang and Yoo [77] extended this to Bidirectional LSTM (Bi-LSTM) for Kubernetes auto-scaling, achieving RMSE 1.31× smaller than ARIMA and R^2 = 0.9954 on the FIFA World Cup trace. Their system incorporated a 60-second cooling-down period and a Resource Removal Strategy to mitigate oscillation during scale-in. The HANSEL framework [78] further advanced Bi-LSTM-based horizontal scaling with adaptive microservice-level prediction and SLA compliance enforcement.

2. **Temporal Convolutional Networks.**

   Dang-Quang and Yoo [66] proposed TCN-based proactive auto-scaling that balances resource efficiency, cost, and SLA compliance by managing three metrics simultaneously. TCNs use dilated causal convolutions to capture long-range patterns without the sequential computation constraint of RNNs.

3. **Graph Neural Networks.**

   The Graph-PHPA framework [79] combined LSTM for temporal prediction with Graph Neural Networks (GNN) to model microservice dependency graphs. By capturing inter-service relationships, Graph-PHPA achieved superior resource savings over standard Kubernetes HPA for the Bookinfo application. DeepScaler [67] extended this concept with spatiotemporal GNN and adaptive graph learning for holistic microservice auto-scaling.

4. **Sequence-to-Sequence Models.**

   Luo et al. [80] developed the Madu microservice autoscaler using Seq2Seq models with a custom loss function incorporating data-dependent uncertainty. Their stochastic attention mechanism achieved 92.3% prediction accuracy (13% higher than prior state-of-the-art) on Alibaba Cluster Traces and DeathStarBench, with 1.7× resource reduction compared to reactive auto-scaling.

2.4.2.4 Transformer-Based Architectures

The attention mechanism introduced by Vaswani et al. [81] has been adapted for time-series forecasting. Kumar et al. [82] developed InformerAutoScale using the ProbSparse attention mechanism of the Informer architecture, achieving a 90.66% scaling efficiency improvement over reactive HPA and RMSE = 3.03 (versus ARIMA's 14.03) on NASA HTTP traces.

Kumar and Verma [83] proposed a Multivariate Transformer-based MAPE auto-scaling framework that uses Pearson correlation for multivariate feature selection and achieves an elastic speedup of 2.9818 on the Bitbrains GWA-T-12 dataset. Bâ et al. [84] leveraged Temporal Fusion Transformer (TFT) interpretability to directly translate prediction feature importance into actionable scaling decisions, bridging the gap between prediction and execution.

TABLE 2.5: Summary of hybrid auto-scaling studies

Ref.	Technique	Scaling Metric	Scaling Type	Evaluation Metrics	Key Limitation
[85]	Control, DL-based forecasting	Resource utilization	Horizontal	Utilization, SLA violation, cost	Optimization overhead may increase complexity.
[64]	Control, ML-based forecasting	Utilization, response time	Horizontal	SLA violation, utilization	Bursty workloads may reduce prediction accuracy.
[78]	Control, Bi-LSTM, RL	Utilization, queue length, response time	Horizontal	SLA violation, utilization	Abnormal traffic fluctuations remain difficult to handle.
[86]	Control, statistical forecasting	Utilization, request completion time	Horizontal	Processing time, utilization	Depends on reactive behaviour while collecting historical data.
[87]	Control, statistical forecasting	Response time, throughput	Horizontal	SLA violation, over-/under-provisioning time	Non-stationary workloads remain challenging.
[56]	RL, fuzzy logic	Utilization, response time, instances	Horizontal	Response time, latency, cost	Inefficient scaling may occur during initial training.
[41]	Control, statistical forecasting	Utilization, throttling probability	Vertical	Utilization, throttling, cost, SLA violation	Rolling updates may introduce delay.
[38]	Control, queuing, forecasting	Request rate, utilization	Horizontal	Utilization, SLO violation, response time, cost	Queuing model requires application-specific tuning.

2.4.2.5 Reinforcement Learning Approaches

Reinforcement Learning (RL) frames auto-scaling as a Markov Decision Process where an agent learns optimal scaling policies through interaction with the environment.

Lipari et al. [88] combined LSTM workload forecasting with a Deep Q-Network (DQN) for Kubernetes auto-scaling, achieving up to 10% resource reduction and 30% performance improvement over HPA. Wanigasooriya and Ekanayake [89] developed NimbusGuard, augmenting DQN with LSTM workload forecasts for proactive Kubernetes auto-scaling with superior cost efficiency.

The PPO-based approach of [73] applied Proximal Policy Optimisation for horizontal pod auto-scaling, while [90] optimised cloud workloads using model-free RL. Ma et al. [91] used MDP-based RL to provide future state prediction and resource optimisation for container scaling. The AWARE system [74] deployed RL-based workload auto-scaling in production cloud systems, using batch RL with conservative policy updates to prevent performance degradation during exploration.

Meng et al. [92] proposed BAScaler, a multi-level ML approach combining burst detection, overestimation for bursts, and RL correction for non-bursts, achieving 57% average SLO violation reduction across ten real-world workloads.

Despite promising results, RL-based auto-scaling faces challenges: training requires substantial environment interaction that may cause SLA violations during exploration, and learned policies may not generalise across workload distributions [88, 74]. Several studies have attempted to overcome the weakness of single-model workload prediction by combining, selecting, or weighting multiple predictors. These works differ in how the combination is performed. Some approaches select one predictor based on workload characteristics, some combine predictions through dynamic weights, and others build hybrid models in which one model explains the residual error of another. The following discussion reviews these ensemble and hybrid approaches and identifies their relevance to proactive auto-scaling.

2.4.2.6 Ensemble Methods for Workload Prediction

Ensemble methods combine multiple base predictors to leverage their complementary strengths and mitigate individual weaknesses [93, 94]. This is particularly relevant for cloud workload prediction, where no single model dominates across all workload patterns.

Saxena et al. [48] classified machine-learning-based cloud workload prediction models into evolutionary, deep learning, hybrid, ensemble, and quantum learning categories, and showed through comparative evaluation that ensemble learning is one of the major approaches for improving workload prediction accuracy.

Shariffdeen et al. [60] demonstrated the effectiveness of ensemble forecasting for PaaS auto-scaling by combining ARIMA, exponential smoothing, and regression. Their inteliScaler system used static weighting based on validation-set performance. Singh et al. [95] also emphasized the limitation of single-model prediction by stating that no prediction model is generally best for all workload patterns. Their TASM approach combines Linear Regression, ARIMA, and SVR through a workload classifier that selects the most suitable model according to the characteristics of the current workload.

Nikhil and Kumar [96] proposed a more sophisticated ensemble combining Linear Regression, ARIMA, SVM, and Neural Networks with an SVM-based dynamic weighting mechanism that adapts predictor contributions based on recent error history. Toka et al. [97] developed a competing multi-forecast scaling engine for Kubernetes edge clusters, where multiple time-series models compete and the best performer for the current window is dynamically selected.

Nguyen et al. [94] proposed an ensemble combining multiple base learners for cloud workload prediction and demonstrated consistent accuracy improvements over individual models. Pandeeswari and Kumar [98] extended ensemble methods with adaptive fusion mechanisms that account for workload regime transitions. In their approach, ARIMA captures the linear component, LSTM learns nonlinear residual behaviour, and ELM contributes to the final ensemble prediction. The proposed model outperformed Prophet and standalone ARIMA–LSTM across all reported metrics. Similarly, Guruge and Priyadarshana [99] proposed a Prophet–LSTM hybrid model for proactive Kubernetes auto-scaling. On the NASA dataset, their hybrid model achieved 69.4% lower MSE and 78.4% lower MAE than the best single-model Bi-LSTM baseline. This result shows the practical value of combining a seasonality-aware statistical model with a neural model capable of learning nonlinear residuals.

2.4.2.7 Clustering-Enhanced Ensemble Approaches

A key innovation in ensemble workload prediction is the use of *clustering* to characterise workload patterns and assign appropriate predictor weights. Iqbal et al. [62] pioneered this concept by using K-means and EM clustering to partition web request patterns, then applying specialised predictors to each cluster.

MacQueen's K-means algorithm [100] partitions workload windows into clusters based on statistical features (variance, periodicity, and rate of change). By computing the proximity of the current observation window to each cluster centroid, model-specific weights can be dynamically assigned, giving higher weight to the model that historically performs best on the identified pattern type.

Hence, the core challenge in cloud computing is accurately forecasting short-term workload metrics, such as request rates, within a 10–30 minute horizon to enable timely resource scaling, minimize latency, and prevent system overload. However, cloud workloads are highly dynamic, exhibiting a mixture of linear trends, seasonal patterns, and sudden nonlinear spikes driven by user behavior, multi-tenancy, and evolving computing paradigms, making prediction inherently difficult [101]. Traditional statistical approaches like ARIMA and exponential smoothing struggle with such non-stationary and complex patterns, particularly failing to capture abrupt changes and long-term dependencies [61]. While machine learning and deep learning models, including LSTM, RNN, and Transformer-based methods, improve the ability to model sequential and nonlinear relationships, they introduce challenges such as high computational cost, risk of overfitting or local optima, and limited suitability for real-time deployment [102, 80]. Hybrid and ensemble approaches attempt to combine strengths of multiple models but often rely on static weighting, limited feature decomposition, or lack adaptability to concept drift and multi-scale temporal dynamics. Additionally, uncertainty-aware and clustering-based methods provide useful insights but frequently lack real-time adaptability and integration with scalable cloud systems. These limitations highlight a clear research gap for a flexible, adaptive ensemble framework that can dynamically learn multi-scale patterns, adjust to evolving workloads, and provide accurate, real-time predictions suitable for modern cloud environments.

The present thesis (Chapter 3) extends this concept by combining K-means clustering with a Prophet + LSTM + Linear Regression ensemble under a dynamic weighted fusion mechanism, achieving state-of-the-art prediction accuracy on the NASA and ClarkNet HTTP traces.

2.4.3 Comparative Summary of Proactive and Hybrid Approaches

Table 2.6 provides a comparative summary of the major categories of proactive and hybrid auto-scaling approaches discussed in the literature, highlighting their representative techniques, main advantages, and key limitations.

TABLE 2.6: Comparative Summary of Existing Proactive and Hybrid Auto-Scaling Approaches

Category	**Representative Techniques**	**Advantages**	**Limitations**
Statistical Time-Series	ARIMA, SARIMA, Holt-Winters, Prophet	Simple, interpretable, low computational cost; effective for seasonal patterns	Limited ability to capture non-linear and bursty behaviour; weak under regime changes
Classical Machine Learning	Linear Regression, SVR, Random Forest, Decision Trees	Faster training than deep models; reasonable accuracy with engineered features	Requires careful feature engineering; struggles with long-term temporal dependencies
Deep Learning (Single Model)	LSTM, Bi-LSTM, GRU, TCN, Seq2Seq	Strong capability to model non-linear and long-range temporal patterns	High training cost, risk of overfitting, slower inference, and limited interpretability
Ensemble / Hybrid Forecasting	Prophet+LSTM, ARIMA+LSTM, weighted multi-model ensembles	Higher accuracy by combining complementary models; more robust across workload types	Increased complexity and computational overhead; weight tuning can be non-trivial
Reinforcement Learning	DQN, PPO, A3C, Actor-Critic	Learns adaptive scaling policies through interaction; can optimise long-term rewards	Requires extensive training data/simulation; slow convergence; sensitive to reward design
Hybrid (Prediction + RL / Reactive)	LSTM+PPO, Prediction + Threshold, Proactive + Reactive controllers	Combines foresight of prediction with adaptability of RL or safety of reactive rules	Higher system complexity; coordination between components needs careful design
Proposed Approach	Clustering-enhanced Hybrid Ensemble (K-means + Prophet + LSTM + LR) with stability controls	Pattern-aware prediction, improved accuracy across diverse workloads, and practical stability mechanisms	Currently evaluated mainly on CPU-intensive workloads and Docker Swarm environment

From the comparison, it is evident that most existing proactive methods rely on a

single forecasting model, which performs well only on specific workload patterns. Ensemble and hybrid approaches generally achieve better prediction accuracy but often increase system complexity. Moreover, very few works combine *pattern-aware ensemble prediction* with practical stability mechanisms (such as smoothing, cooldown, and safety margins) and evaluate them on a real container orchestration platform. These observations directly motivate the clustering-enhanced hybrid ensemble model and the stability-aware proactive auto-scaling framework proposed in this thesis.

2.4.4 Research Gap

Analysis of the surveyed literature reveals the following critical gaps that motivate the proposed research:

1. **Reactive Scaling Lag in Container Environments.** Most production systems, including Kubernetes HPA [20], rely on infrastructure-level monitoring metrics such as CPU-utilization thresholds. Since container startup takes 30–90 seconds, reactive systems consistently under-provision during demand spikes, causing SLA violations.

2. **Lack of Accurate Prediction-Based Proactive Scaling for Multi-Pattern Workloads.** Existing predictive auto-scalers use a single model (LSTM, ARIMA, or Prophet), which performs well only on specific workload patterns. Real-world workloads exhibit mixed seasonality, burstiness, and steady trends simultaneously. No prior work combines K-means workload clustering with an inverse-error-weighted ensemble of Prophet, LSTM, and linear regression for auto-scaling.

3. **Lack of Integration with Real Deployment Systems.** Many studies validate prediction accuracy in isolation or use simulation. There is limited work that integrates prediction, monitoring, planning, and scaling execution in a real container orchestration testbed such as Docker Swarm.

4. **Insufficient Integration of Workload-Pattern-Aware Prediction with Real-Time System Constraints.** A practical auto-scaler must translate predicted workload into replica count using per-replica capacity while considering current CPU,

latency, replica bounds, and cooldown. This integration is still insufficiently addressed.

2.5 Summary

This chapter has reviewed the state-of-the-art in cloud auto-scaling across six major themes: theoretical foundations of cloud computing and deployment models, auto-scaling taxonomy, reactive techniques, proactive prediction-based approaches, resource placement optimization, and the MAPE-K control loop.

Key findings from the literature are as follows. Reactive auto-scaling methods such as Kubernetes HPA remain the production standard but suffer from scaling lag that leads to SLA violations during demand surges. Proactive approaches using deep learning models (LSTM, Bi-LSTM, TCN, and Transformers) consistently outperform reactive baselines; however, these models are computationally complex, require large amounts of training data, and are difficult to interpret. Ensemble and clustering-enhanced prediction methods achieve superior accuracy across diverse workload patterns compared to single models. The MAPE-K architecture provides a structured framework for autonomic scaling, yet few studies implement all phases with comprehensive stability controls.

Despite these advances, critical gaps remain in integrating pattern-aware ensemble prediction with real-time stability mechanisms, and in thorough evaluation on real container orchestration platforms using public traces. The subsequent chapters of this thesis address these gaps through a clustering-enhanced hybrid ensemble workload prediction model (Chapter 3), a MAPE-K-based proactive auto-scaling framework with stability controls (Chapter 4), its implementation on a Docker Swarm testbed (Chapter 5), and systematic evaluation across multiple workload patterns (Chapter 6).

Chapter 3

Hybrid Ensemble Workload Prediction Model

3.1 Overview

Accurate workload prediction is the cornerstone of proactive auto-scaling. A predictor that underestimates future demand causes under-provisioning, which leads to SLA violations; one that overestimates causes over-provisioning, which wastes resources and increases operational cost. The challenge is that real web-server workloads are non-stationary, combining periodic daily cycles, monotonic trends, non-linear bursts, and irregular fluctuations within a single trace.

This chapter presents the clustering-enhanced hybrid ensemble prediction model. The model addresses the multi-pattern challenge by first identifying the current workload regime through K-means clustering and then assigning dynamic weights to three base predictors: Facebook Prophet , Long Short-Term Memory (LSTM) networks, and Linear Regression. The weighted ensemble output is used as the predicted workload $\hat{A}(t+H)$ in the MAPE-K control loop described in Chapter 4.

The chapter is organized as follows. Section 3.2 formally defines the prediction problem. Sections 3.3 and 3.4 describe the datasets and the preprocessing pipeline. Section 3.5 defines the workload feature space used for clustering. Section 3.6 and Section 3.7 describe the K-means clustering and workload classification procedure. Section 3.8 describes each base model. Section 3.9 derives the dynamic weight assignment scheme. Also summaries the complete prediction pipeline. Section 3.10 gives the time and space complexity of the proposed hybrid model. Section 3.11 presents the experimental results. Section 3.12 summaries the chapter.

3.2 Problem Formulation

Let $\{x_t\}_{t=1}^{T}$ denote a univariate time series, where $x_t \in \mathbb{R}$ represents the observed workload (e.g., request rate) at discrete time step t, and T is the total length of the series.

The goal is to learn a parametric predictor f_θ (parameterized by θ) that maps a historical window of length ω to an H-step-ahead forecast. Specifically, at each time t, the input window is defined as

$$W_t = \{x_{t-\omega+1}, \ldots, x_t\}, \tag{3.1}$$

where ω is the look-back window size (number of past observations used as input).

The predictor produces an H-step forecast

$$\widehat{y_t} = f_\theta(W_t) = (\widehat{x_{t+1}}, \widehat{x_{t+2}}, \ldots, \widehat{x_{t+H}}), \tag{3.2}$$

where H is the forecast horizon (number of future time steps to predict) and $\widehat{x_{t+h}}$ denotes the predicted value at step $t+h$.

The model is trained by minimizing the empirical risk over all admissible windows

$$t \in \{\omega, \omega+1, \ldots, T-H\}, \tag{3.3}$$

using a direct multi-output strategy and the squared error loss, with appropriate regularization to improve generalization.

Figure 3.1 illustrates the recursive multi-step forecasting approach, in which previously predicted values are iteratively fed back as inputs for subsequent steps.

A carefully chosen window size ω and forecast horizon H are critical. An overly short window or an inappropriate horizon can lead to oscillations in resource provisioning, whereas an optimally selected pair enables accurate trend capture and stable predictions [47].

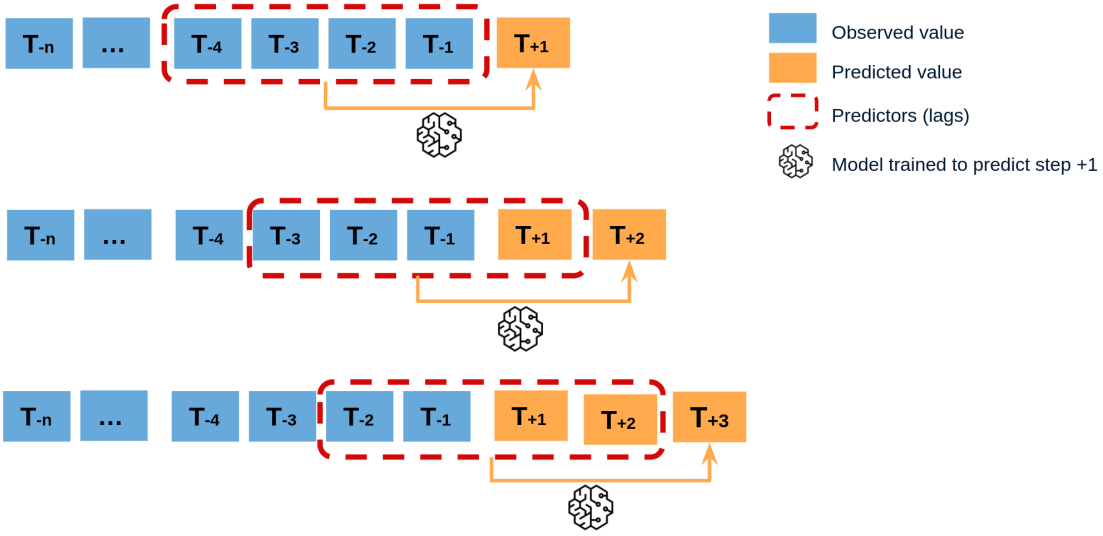


FIGURE 3.1: Diagram of recursive multi-step forecasting

3.3 Datasets

3.3.1 NASA HTTP Logs

The NASA HTTP dataset [103] records all HTTP requests to the Kennedy Space Centre web server from 1 July to 31 August 1995, comprising approximately 3.46 million requests. The raw log is parsed to extract request timestamps, which are then aggregated into one-minute intervals to produce a time series of request counts. The maintenance gap between the two months is removed, yielding 44,640 samples spanning 31 effective days. The request data shows an average (mean) of 41.23 requests, with a median of 32, a standard deviation of 28.45, and values ranging from a minimum of 0 to a maximum of 248. The workload exhibits strong daily periodicity with a peak-to-trough ratio exceeding 10:1, driven by the daily activity cycle of web users. Figure 3.2 illustrates the trend of NASA HTTP requests over time, indicating changes in request intensity across different time intervals.

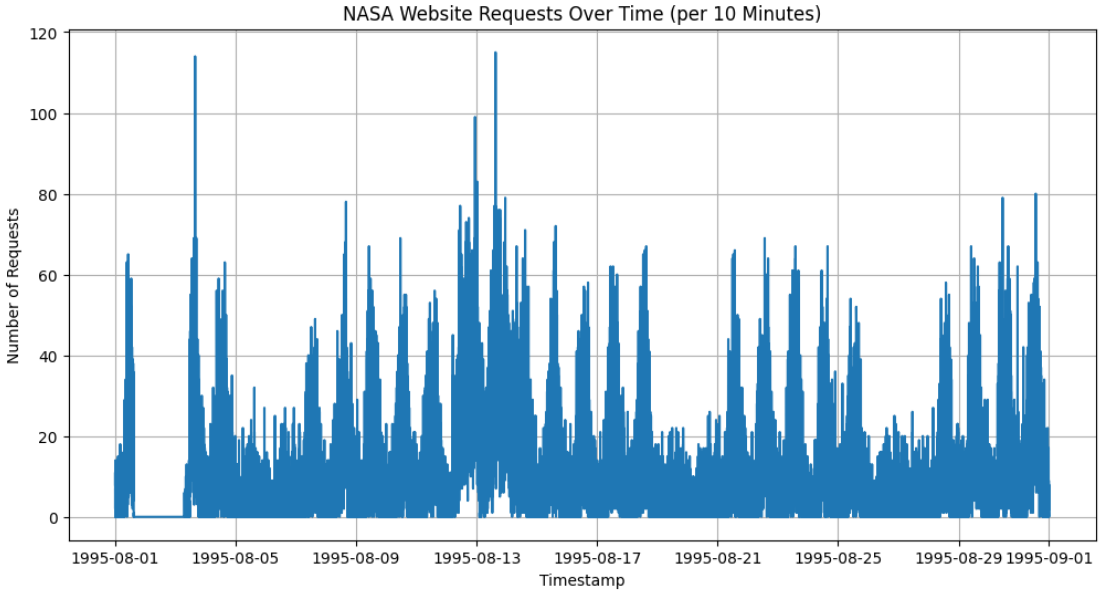


FIGURE 3.2: NASA HTTP requests over time

3.3.2 ClarkNet WWW Logs

The ClarkNet dataset [104] records HTTP requests to the ClarkNet internet service provider from 28 August to 3 September 1995, comprising approximately 3.33 million requests. After aggregation to one-minute intervals, the dataset yields 8,899 samples over seven days. The request data has an average (mean) of 52.67 requests, with a median of 45, a standard deviation of 35.12, and values ranging from a minimum of 0 to a maximum of 312. ClarkNet workloads are more irregular than NASA, with shorter periodic cycles and more pronounced non-linear bursts, providing a complementary evaluation scenario that tests the model's adaptability to less structured patterns. Figure 3.3 illustrates the trend of Clarknet HTTP requests over time, indicating changes in request intensity across different time intervals.

3.3.3 Quantitative Characteristics of the NASA and ClarkNet Datasets

Figure 3.4 illustrates the ACF and PACF plots for the NASA and ClarkNet datasets (10-minute intervals), used to identify autocorrelation patterns and determine suitable model parameters. Figure 3.5 illustrates the distribution of website requests per hour for the ClarkNet and NASA datasets, highlighting temporal usage patterns and variability.

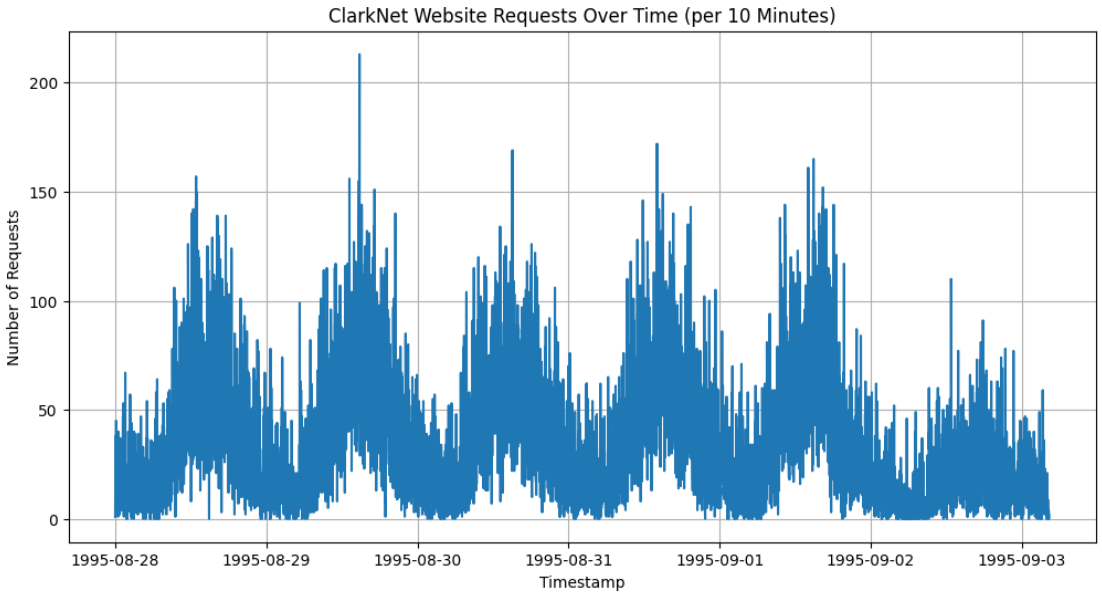
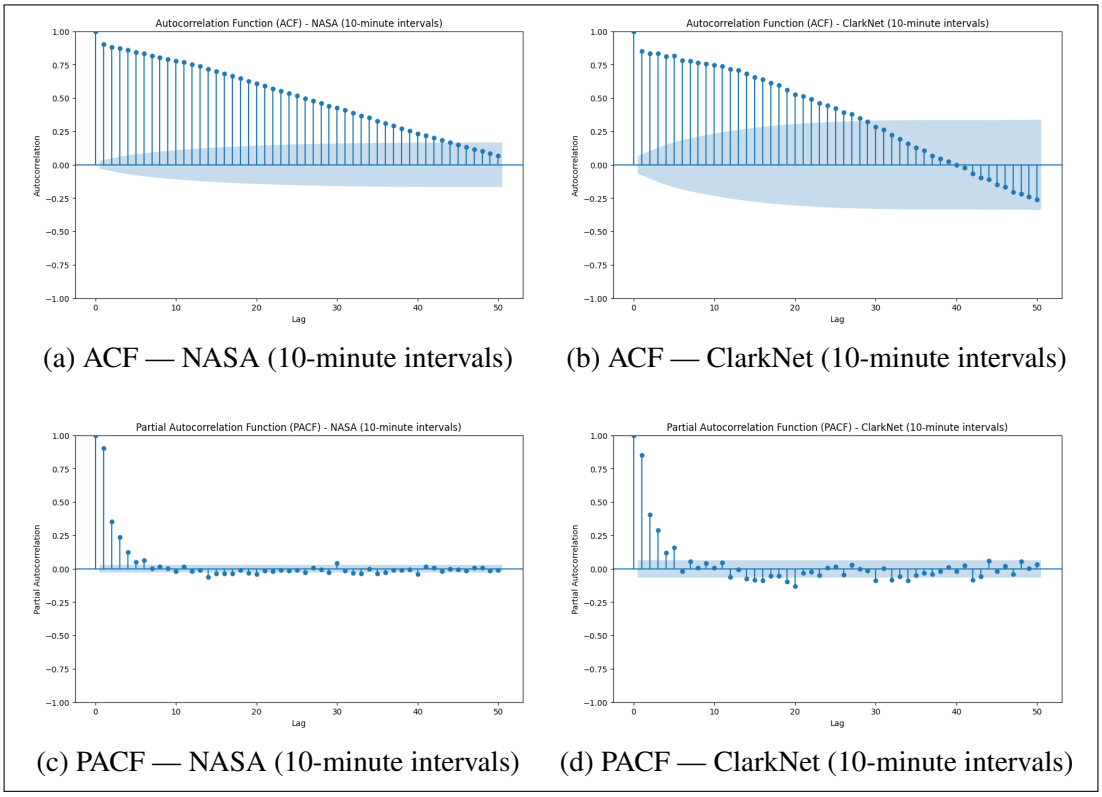


FIGURE 3.3: Clarknet HTTP requests over time

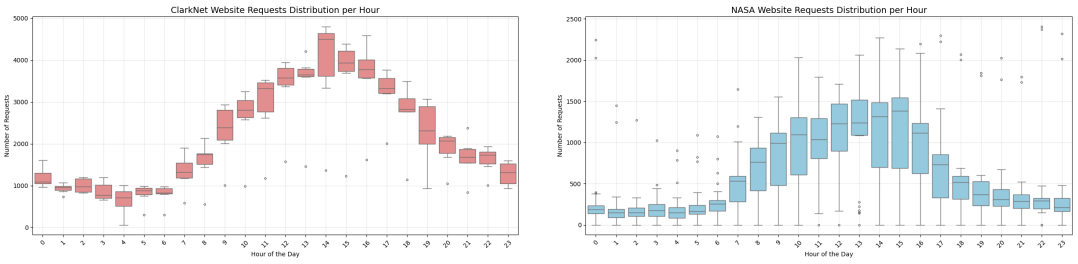
The ACF and PACF analysis confirms that the NASA and ClarkNet workloads are not random but temporally correlated. Strong autocorrelation at the initial lags indicates that recent workload observations are important for forecasting. The PACF plots show large spikes at the first few lags, especially lag 1 to lag 3, after which the coefficients quickly approach zero. This suggests that short-term history has the strongest direct influence on current workload. In ClarkNet, the ACF becomes negative after approximately lag 40, indicating possible cyclic or oscillatory traffic behavior. These observations justify the use of time-series and sequence-based forecasting models such as ARIMA/SARIMA, LSTM, Prophet, and the proposed hybrid ensemble approach.



(a) ACF — NASA (10-minute intervals) (b) ACF — ClarkNet (10-minute intervals)

(c) PACF — NASA (10-minute intervals) (d) PACF — ClarkNet (10-minute intervals)

FIGURE 3.4: ACF and PACF plots for the NASA and ClarkNet datasets (10-minute intervals).



(a) ClarkNet (b) NASA

FIGURE 3.5: Website requests distribution per hour for the ClarkNet and NASA datasets.

3.4 Data Pre-processing

Data pre-processing is the foundational step in the model training, transforming raw workload logs into a structured format suitable for predictive modeling. Cloud workloads, such as HTTP request rates from NASA and ClarkNet datasets, are inherently noisy, with missing values, outliers, and irregular sampling intervals due to the nature of system logs. To focus on valid loads, the process begins by parsing the log files to

extract timestamped request counts and filtering successful requests (`status` < 400). Aggregation occurs at a resolution of 1 minute using grouping functions. To ensure temporal fidelity and mitigate noise, linear interpolation is employed for missing values, and outlier capping with a 3-sigma threshold is applied during pre-processing, thereby avoiding the over-regularization risks associated with methods such as Savitzky-Golay filtering.

A multi-step transformation is used to address non-stationarity, a prevalent issue in cloud data where mean and variance vary over time. First, a log transformation is applied:

$$x'_t = \log(1 + x_t), \tag{3.4}$$

which stabilizes variance and manages zeros, preventing division-by-zero errors in subsequent calculations. This transformation approximates normality and reduces the influence of extreme values, making it effective for positively skewed distributions.

Next, seasonal differencing with a lag of 1440 minutes (daily cycle) removes periodic trends:

$$x''_t = x'_t - x'_{t-1440}, \tag{3.5}$$

Finally, first-order differencing eliminates residual trends:

$$x'''_t = x''_t - x''_{t-1}, \tag{3.6}$$

Stationarity is validated using the Augmented Dickey-Fuller (ADF) test ($p < 0.05$ for stationarity) and the Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test. Outliers are capped using a z-score threshold $|z| > 3$.

To prepare for clustering, extracted features are normalized to the range $[0, 1]$ using min–max scaling:

$$x_t^{\text{norm}} = \frac{x_t - x_{\min}}{x_{\max} - x_{\min}}. \tag{3.7}$$

The pre-processing pipeline slightly reduces data length while preserving temporal integrity. The processed time series x_t is then ready for feature extraction.

3.5 Feature Extraction

Feature extraction transforms the pre-processed time series into a compact representation that captures workload dynamics for clustering and classification. A sliding window of size ω minutes is used to compute three features.

3.5.1 Variance

The Variance (Var_t) measures the volatility of workload:

$$\text{Var}_t = \frac{1}{h-1} \sum_{i=t-h+1}^{t} (\text{requests}_i - r_t)^2, \tag{3.8}$$

where the window mean is

$$r_t = \frac{1}{h} \sum_{i=t-h+1}^{t} \text{requests}_i. \tag{3.9}$$

3.5.2 Periodicity

Periodicity (Per_t) quantifies seasonal strength using the autocorrelation function (ACF):

$$\text{Per}_t = \max_{1 \le k \le p} \frac{\sum_{i=t-h+1}^{t-k} (\text{requests}_i - r_t)(\text{requests}_{i+k} - r_t)}{\sum_{i=t-h+1}^{t} (\text{requests}_i - r_t)^2}, \tag{3.10}$$

where $p = 1440$, corresponding to daily seasonality.

3.5.3 Rate of Change

Rate of Change (Roc_t) captures workload trends:

$$\text{Roc}_t = \frac{\text{requests}_t - \text{requests}_{t-1}}{\text{requests}_{t-1} + \epsilon}, \tag{3.11}$$

where $\epsilon = 0.01$ prevents division by zero.

The features are standardized as follows.

$$x_t^{\text{std}} = \frac{x_t - \mu_x}{\sigma_x}, \tag{3.12}$$

where μ_x and σ_x denote the mean and standard deviation of feature vectors.

The resulting feature vector is

$$\mathbf{f}_t = \begin{bmatrix} \text{Var}_t \\ \text{Per}_t \\ \text{Roc}_t \end{bmatrix}. \tag{3.13}$$

3.6 Workload Clustering

Workload clustering groups time steps into distinct categories such as seasonal, non-linear, and steady using K-means, an unsupervised method that partitions data by minimizing intra-cluster variance. We set $K = 3$, as empirical analysis shows, three clusters adequately capture the diversity in datasets. The objective function is

$$J = \sum_{i=1}^{K} \sum_{\mathbf{x}_t^{\text{std}} \in S_i} \left\| \mathbf{x}_t^{\text{std}} - \mathbf{c}_i \right\|^2, \tag{3.14}$$

where S_i the set of feature vectors in cluster i and $\mathbf{c}_i$ is the centroid of cluster i. The term $\mathbf{x}_t^{\text{std}} - \mathbf{c}_i$ computes the Euclidean distance from a feature vector to each centroid, and the algorithm iterates until convergence. Cluster interpretation is post-hoc: centroids with high periodicity correspond to seasonal patterns, high variance/rate of change to non-linear ones, and low values to steady ones. Validity is assessed with a silhouette score (0.5–0.7 in our data), ensuring well-separated clusters. Figure 3.6 illustrates K-means clustering (K=3) of NASA workload patterns. Figure 3.7 illustrates K-means clustering (K=3) of ClarkNet workload patterns.

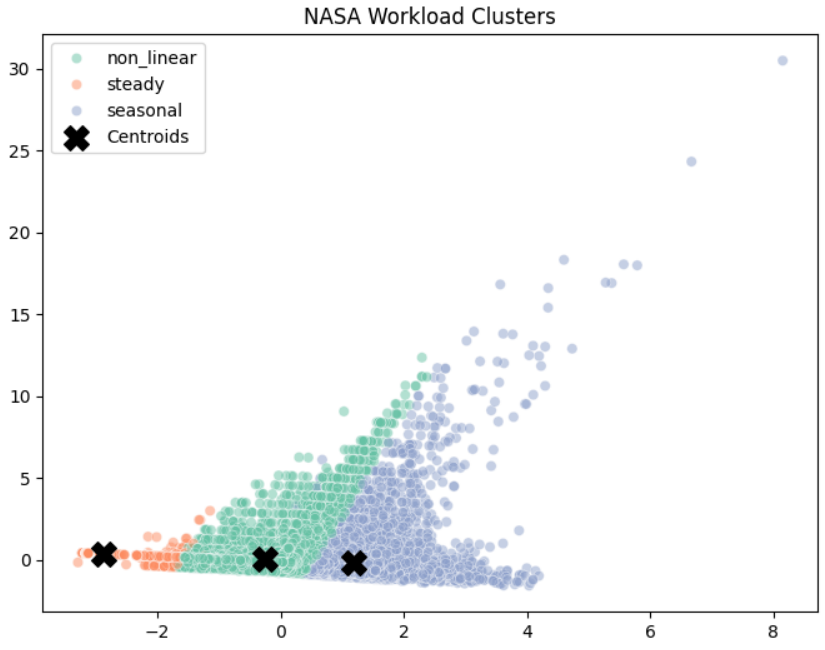


FIGURE 3.6: NASA Cluster

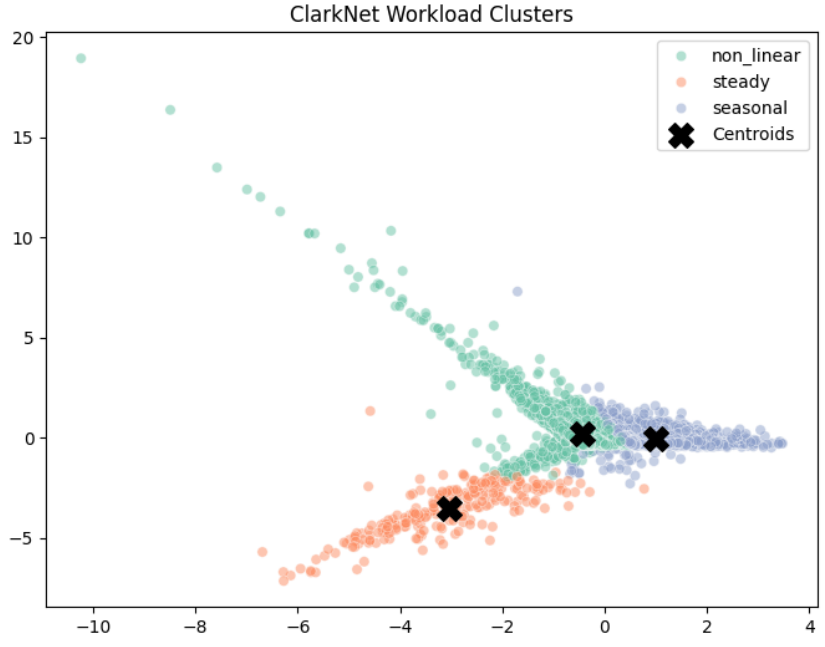


FIGURE 3.7: Clarknet cluster

3.7 Workload Classification

Workload classification assigns dynamic weights to prediction models based on proximity to cluster centroids, allowing soft, adaptive selection without rigid boundaries. This step builds on clustering by quantifying how closely a time step's features match each pattern, allowing the model to blend contributions proportionally.

For each time step t, distances are calculated as

$$d_i(t) = \left\| \mathbf{x}_t^{\text{std}} - \mathbf{c}_i \right\|^2, \tag{3.15}$$

measuring Euclidean separation from centroid i.

Weight computation ($w_i(t)$). Weights for ensemble prediction are computed based on inverse Euclidean distances to cluster centroids. This ensures that the model closest to the characteristics of the workload (lowest $d_i(t)$, highest $w_i(t)$) contributes more, reducing prediction errors compared to a uniform average. The weights are computed as

$$w_i(t) = \frac{1/(d_i(t) + \epsilon)}{\sum_{j=1}^{3} 1/(d_j(t) + \epsilon)}, \tag{3.16}$$

where $w_1(t)$ is the weight for the seasonal cluster (Prophet), $w_2(t)$ is the weight for the non-linear cluster (LSTM), and $w_3(t)$ is the weight for the steady cluster (LR). Here $\epsilon = 0.01$ ensures numerical stability and $\sum_{j=1}^{3} w_j(t) = 1$. In ambiguous cases, the predictions of multiple models are combined using a weighted ensemble, where the weight of each model is proportional to the inverse of its validation RMSE, ensuring that models with lower error contribute more strongly to the final forecast.

The classification is efficient ($O(n * K),\ K = 3$) and generic, avoiding fixed thresholds to maintain adaptability across datasets. It extends static classification by providing continuous probabilities, handling mixed workloads where a time step may exhibit partial seasonality and non-linearity. This approach mitigates overfitting in single models and improves generalization, as weights evolve with data shifts.

3.8 Prediction Models

The prediction models form the core of the ensemble; each tailored to a workload pattern for specialized forecasting.

Prophet is a time-series forecasting model designed to capture trend and seasonality using an additive decomposition framework. Prophet handles seasonal patterns with an additive model

$$P_{\text{Prophet}}(t) = g(t) + s(t) + \epsilon_t, \tag{3.17}$$

where $g(t)$ is the trend (piecewise linear), $s(t)$ is seasonality (Fourier series for daily cycles), and ϵ_t is error. This model is inspired by the architecture described in [105].

Long Short-Term Memory (LSTM) [75] captures non-linear dynamics with a recurrent architecture: input layer, two LSTM layers, and dense output. Gates manage the information flow:

$$\begin{aligned} f_t &= \sigma(W_f[h_{t-1}, x_t] + b_f), \\ i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i), \\ o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o), \\ c_t &= f_t c_{t-1} + i_t \tanh(W_c[h_{t-1}, x_t] + b_c), \\ h_t &= o_t \tanh(c_t). \end{aligned} \tag{3.18}$$

Here f_t, i_t and o_t denote the forget, input, and output gates respectively, which regulate the flow of information. W_f, W_i, W_o and W_c are the weight matrices, while b_f, b_i, b_o and b_c are the bias terms. The cell state c_t preserves long-term memory, and the hidden state h_t captures the output representation at time t as shown in Figure 3.8. This design is inspired by the architecture described in [75].

Linear Regression (LR) fits steady patterns using least squares on lagged inputs:

$$P_{\text{LR}}(t) = \beta_0 + \sum_{i=1}^{w} \beta_i x_{t-i}, \tag{3.19}$$

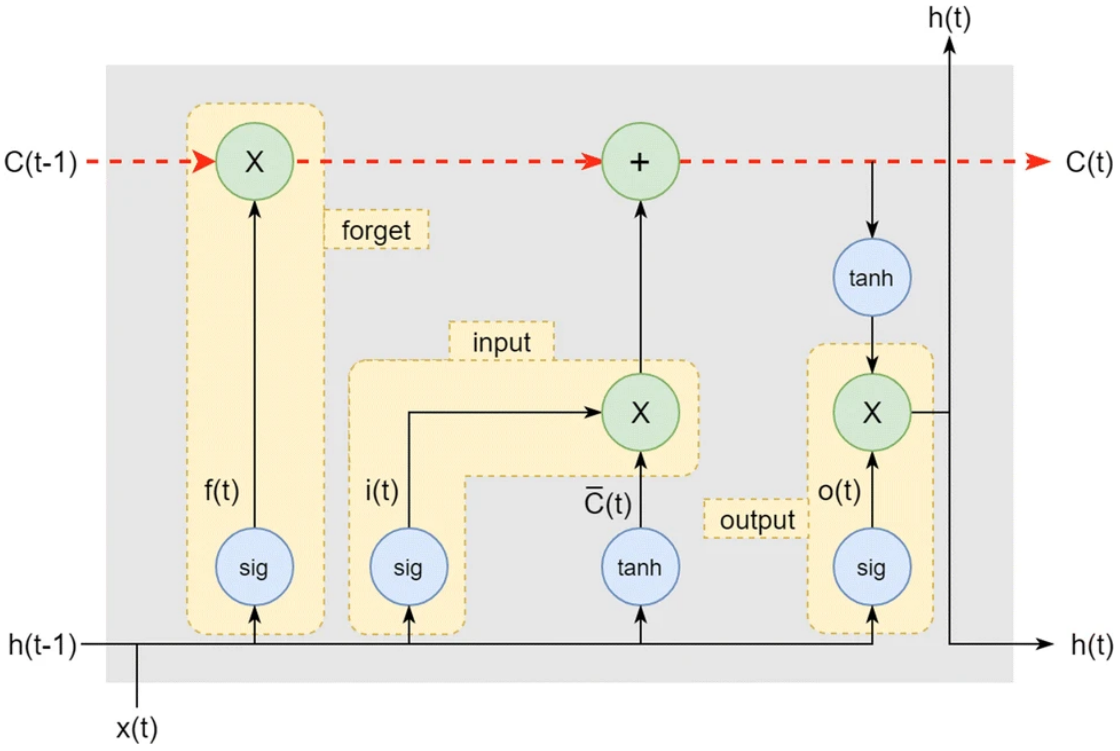


FIGURE 3.8: LSTM Architecture [75]

where β_0 represents the intercept, β_i are the regression coefficients, w denotes the window size (number of lagged inputs), and x_{t-i} indicates the past workload values. Linear Regression is chosen in this study as it effectively captures and predicts steady workload patterns. Its ability to model linear dependencies in historical request trends makes it suitable for stable workload forecasting. This tri-model selection covers diversity, with training efficiency.

3.9 Ensemble Prediction

The performance differences among Linear Regression (LR), Prophet, and Long Short-Term Memory (LSTM) stem fundamentally from their architectural design and training paradigms, which make each model optimal only under specific workload characteristics. LR employs a simple linear formulation with closed-form training, enabling high efficiency and strong performance on stable workloads, but it lacks the representational capacity to capture nonlinear and time-varying dynamics. Figure 3.9 presents the

complete workflow of the proposed hybrid ensemble approach for accurate workload prediction.

Prophet, based on an additive decomposition of trend and seasonality, leverages predefined structural components to model periodic patterns effectively; however, this rigidity limits its adaptability to irregular or rapidly changing workloads. In contrast, LSTM utilizes a recurrent neural architecture trained via backpropagation through time, allowing it to learn complex nonlinear relationships and temporal dependencies, making it suitable for bursty and highly dynamic workloads, albeit at higher computational cost and with susceptibility to overfitting. Nevertheless, despite its flexibility, LSTM lacks an explicit mechanism for encoding seasonality, requiring periodic patterns to be implicitly learned from data, which reduces reliability under long or weak seasonal signals and increases dependence on large training datasets [75, 106]. These limitations are not fully addressed by alternative deep learning approaches: recurrent variants such as GRU inherit similar architectural constraints [107], convolutional models are primarily effective for local temporal feature extraction and fail to capture long-range dependencies [108], and Transformer-based architectures, while powerful, impose significant computational overhead and demand large-scale data for stable training [81]. Collectively, these factors imply that each model performs well only within its "best-case" regime but fails to generalize across mixed or non-stationary workload conditions. To address this, the proposed ensemble model integrates these complementary architectures within a clustering-guided, dynamically weighted framework that combines feature-driven workload classification with adaptive weighting based on cluster proximity and recent prediction error. By performing implicit model selection and blending, the hybrid approach achieves robust performance across steady, seasonal, and nonlinear workload regimes, making it more general and reliable than any individual model operating in isolation.

Ensemble prediction employs a per-window inverse error weighting mechanism that dynamically adjusts model contributions based on recent performance. This approach complements the cluster-based weights derived from the proximity of K-means, addressing the limitation of static weighting by incorporating empirical error feedback. For each prediction window, a rolling buffer of the Root Mean Squared Errors (RMSE) of the last M (default 10) windows is maintained for each model (Prophet, LSTM and Linear

Regression) using a quantitative data structure. The inverse error weight for model k is computed as

$$\text{inv_error}_k = \frac{1}{\text{recent_rmse}_k + \epsilon}, \tag{3.20}$$

where $\epsilon = 0.01$ prevents division by zero, and normalized across models as

$$\text{inv_error_weights}_k = \frac{\text{inv_error}_k}{\sum_j \text{inv_error}_j}. \tag{3.21}$$

These weights are combined with the cluster weights $w_{\text{cluster},k}$ using a regularization parameter α (default 0.5):

$$\text{final_weights}_k = \alpha\, w_{\text{cluster},k} + (1-\alpha)\,\text{inv_error_weights}_k, \tag{3.22}$$

and normalized so that $\sum_k \text{final_weights}_k = 1$. After each prediction, the error buffer is updated with the RMSE of the actual value of the current window versus the predicted value, allowing continuous adaptation.

The ensemble prediction aggregates base model outputs using classification weights:

$$\widehat{y_t} = w_1(t) \cdot P_{\text{Prophet}}(t) + w_2(t) \cdot P_{\text{LSTM}}(t) + w_3(t) \cdot P_{\text{LR}}(t). \tag{3.23}$$

This dynamic blending adapts to patterns, producing the final h-minute mean forecast. It outperforms single models by leveraging strengths, with $O(n)$ complexity for real-time use.

The dynamic weight plots shown in Figure 3.10 and Figure 3.11 indicate that the proposed ensemble adapts its reliance on base predictors according to workload characteristics. For the NASA trace, the Seasonal (Prophet) model receives the highest weight for most time steps, indicating dominant periodic behavior, whereas for the ClarkNet trace, the Non-linear (LSTM) model is more frequently prioritized, reflecting stronger irregular and bursty dynamics. The Steady (LR) model contributes mainly during relatively stable periods, confirming its role as a lightweight predictor for smoother workload segments.

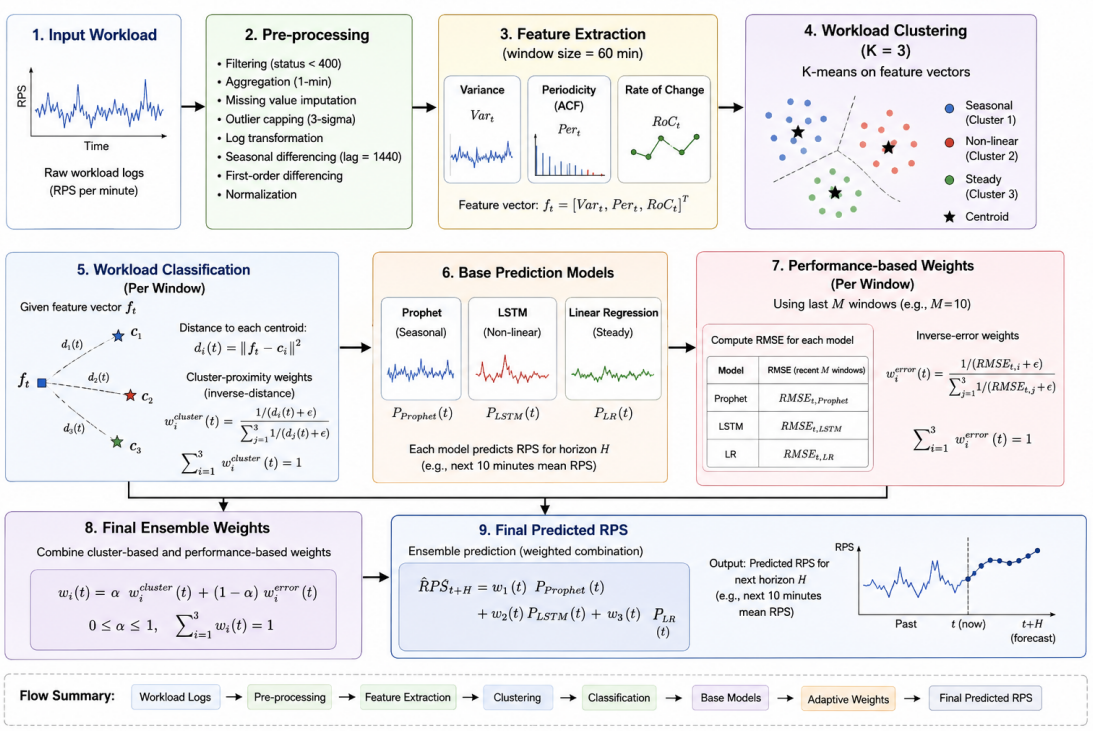


FIGURE 3.9: Hybrid Ensemble Approach for Accurate Workload Prediction

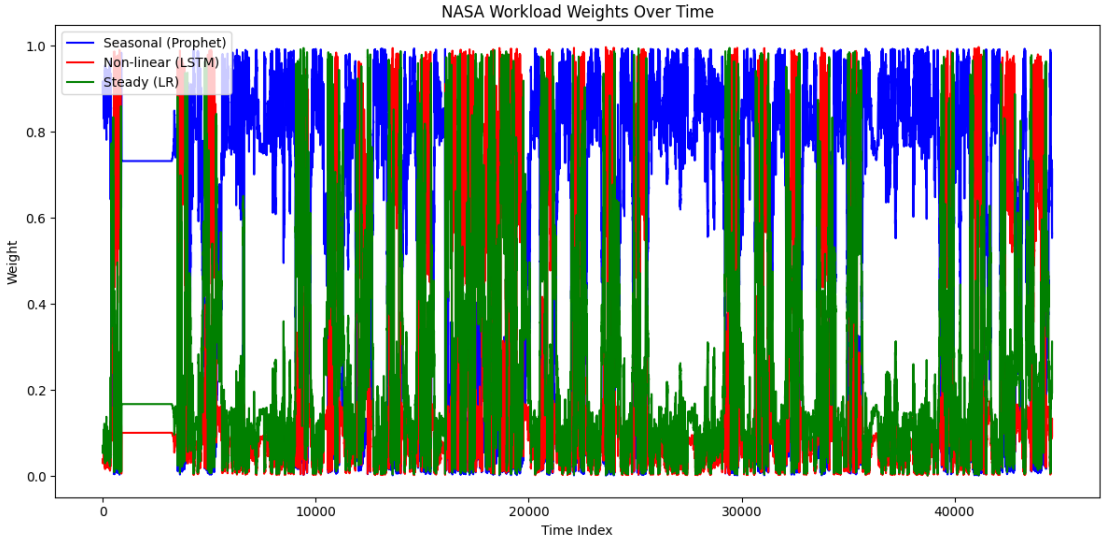


FIGURE 3.10: Workload Weights Over Time - NASA

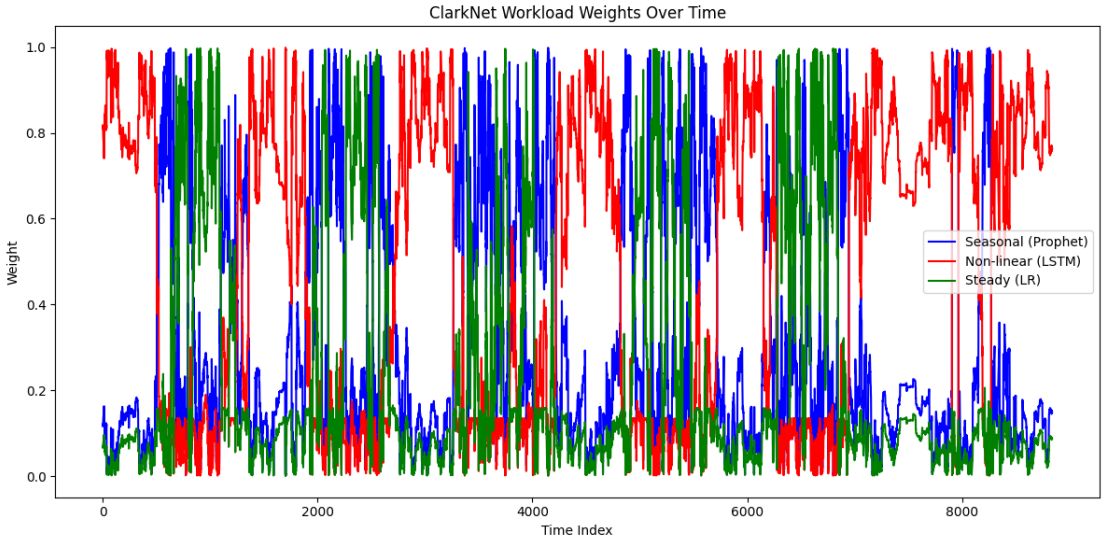


FIGURE 3.11: Workload Weights Over Time - Clarknet

To improve the robustness and adaptability of workload forecasting, the data set was split into training, validation, and test sets (60-20-20) to ensure proper model evaluation. We now include only those base models (Prophet, LSTM, and LR) whose validation RMSE lies within twice the minimum RMSE, thereby filtering out underperforming models. A meta-learner (linear regression) is then trained on the validation predictions, and its coefficients are normalized to obtain dynamic meta-weights, which capture the relative importance of each model.

In parallel, cluster-based weights derived from workload features are integrated, and both sets of weights are combined equally to balance model suitability and empirical performance. In addition, rolling error buffers are introduced to track recent prediction errors and penalize models with deteriorating accuracy, ensuring that the weighting adapts to short-term fluctuations. These modifications make the hybrid approach more selective, adaptive and resilient to workload variations compared to the baseline version.

3.10 Complexity computation

The time and space complexity of the proposed Hybrid model is as follows.

3.10.1 Time Complexity

The time complexity of our hybrid model is dominated by its components during the training and inference phases. Feature extraction over n samples and window size w involves $O(n * w)$ for variance and rate of change, plus $O(n * w * p)$ for periodicity (with p lags for autocorrelation), yielding $O(n * w * p)$ overall. K-means clustering with $K = 3$ clusters and $d = 3$ features is $O(n * K * d * i)$, simplifying to $O(n)$ with fixed K, d, and i iterations. Classification computes inverse-distance weights in $O(n * K)$. Model training adds Prophet's $O(n * \log n)$ for Fourier approximation, LSTM's $O(E * n * h^2)$ with E epochs and h hidden units, and LR's $O(w^2 * n)$. Thus, training complexity is

$$O(n * w * p + n * \log n + E * n * h^2 + w^2 * n). \tag{3.24}$$

Inference, primarily weighting and model predictions, is $O(n * K + n)$, or $O(n)$ with fixed K, making it efficient for real-time use in cloud environments.

3.10.2 Space Complexity

The space complexity of our hybrid model is primarily linear in the number of samples n. Feature storage requires $O(n * d)$ for $d = 3$ features, while centroids use $O(K * d)$ with $K = 3$. Weights occupy $O(n * K)$. Model storage includes Prophet's $O(n)$ for data points, LSTM's $O(h^2)$ for weights, and LR's $O(w)$ for coefficients. Overall space is $O(n)$, as constants like K, d, h, w are fixed, ensuring scalability for large datasets without excessive memory demands.

3.11 Experimental Results

3.11.1 Prediction Accuracy: Model Comparison

All experiments were conducted in a cloud environment equipped with an NVIDIA A100 Tensor Core GPU (40 GB HBM2 memory, ~1.56 TB/s memory bandwidth, Ampere

architecture) and 83 GB system RAM. Minute-wise workload metrics were aggregated into a time series, log-transformed, and differenced (seasonal lag 1440 and first difference) to ensure stationarity (ADF $p < 0.05$). Features (variance, periodicity and rate of change) were extracted over 60-minute windows, standardized, and clustered using K-means ($K = 3$, k-means++ initialization). An ensemble of Prophet, LSTM (2 layers, 50 units, dropout 0.2) and Linear Regression was trained using a 60/20/20 train/validation/test split. The LSTM was trained with Adam (LR = 0.001, 20 epochs), Prophet modelled daily seasonality, and ensemble weights were assigned using inverse-distance weighting based on cluster centroids.

Each dataset was processed in a rolling-forecast manner, iteratively predicting the next forecast horizon ($H = 10$ minutes) mean workload. Baselines included ARIMA, SVR, TCN, LSTM-only, Prophet-only and LR-only (implemented with default settings where applicable for fairness). Hyperparameter tuning was performed using grid search with 5-fold cross-validation to reduce overfitting and improve generalization.

Performance Metrics: The performance of the proposed model is evaluated using standard regression metrics. The Root Mean Squared Error (RMSE) measures the square root of the average squared differences between predicted and actual values, emphasizing larger errors:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(y_t - \hat{y}_t)^2}.$$

The Mean Absolute Error (MAE) computes the average absolute difference, providing a linear measure of error:

$$\text{MAE} = \frac{1}{n}\sum_{t=1}^{n}|y_t - \hat{y}_t|.$$

The Mean Absolute Percentage Error (MAPE) evaluates relative error and is defined as:

$$\text{MAPE} = \frac{100}{n}\sum_{t=1}^{n}\frac{|y_t - \hat{y}_t|}{|y_t| + \epsilon}, \quad \epsilon = 10^{-6}.$$

Finally, the Coefficient of Determination (R^2) indicates the proportion of variance explained by the model:

$$R^2 = 1 - \frac{\sum_{t=1}^{n}(y_t - \hat{y}_t)^2}{\sum_{t=1}^{n}(y_t - \bar{y})^2},$$

where y_t denotes the actual workload, $\hat{y}_t$ the predicted value, $\bar{y}$ the mean of actual values, and n the number of observations. Table 3.1 lists the parameter values used for baseline methods, while Table 3.2 summarizes the parameter settings for the proposed hybrid model.

TABLE 3.1: Parameter values for baseline methods

Baseline model	**Key parameter**	**Value/setting**
ARIMA	Seasonal differencing lag	1440 (daily cycle)
	First differencing lag	1
	Auto-ARIMA seasonal	False
	Seasonal period (m)	1
SVR	Kernel	Radial basis function (rbf)
	Scaler	MinMaxScaler
TCN	Convolutional filters	64
	Kernel size	3
	Optimizer	Adam
	Epochs	10
	Loss function	MSE
LSTM-only	LSTM units (1st layer)	50 (return_sequences=True)
	LSTM units (2nd layer)	50
	Dense units	1
	Optimizer	Adam
	Epochs	10
	Loss function	MSE
Prophet-only	Daily seasonality	True
	Yearly seasonality	False
	Scaler	MinMaxScaler (fitted on y)
	Optimization method	L-BFGS (via CmdStan)
	Iterations	10000
LR-only	Fit method	Ordinary least squares
	Input shape	Flattened (window-based features)

TABLE 3.2: Parameter values for the proposed hybrid model

Component/ parameter	**Key parameter**	**Value/setting**
Feature extraction	Window size (w)	60 minutes
	Maximum periodicity (p)	1440 minutes (daily cycle)
Clustering	Number of clusters (K)	3 (Seasonal, Non-linear, Steady)
	Number of initializations (n_init)	10
	Maximum iterations	300
	Random state	42
Workload classification	Epsilon (inverse distance)	0.01
Per-window adaptive weighting	Rolling window size (M)	10 windows
	Regularization parameter (α)	Meta-learned weights using Linear Regression
	Error metric	RMSE (per window)
Ensemble prediction	Prediction horizon (H)	10 minutes
	Models integrated	Prophet, LSTM, Linear Regression
Training configuration	Train/validation/test split	60% / 20% / 20%
	Optimizer (LSTM)	Adam
	Epochs (LSTM)	20
	Loss function (LSTM)	MSE
Evaluation metrics	MAE, RMSE, MAPE, R^2	Computed post-prediction

Table 3.3 and Table 3.4 present the prediction accuracy of the proposed hybrid ensemble model against six baseline methods on the NASA and ClarkNet datasets, respectively, at $W = 60$ minutes and $H = 10$ minutes.

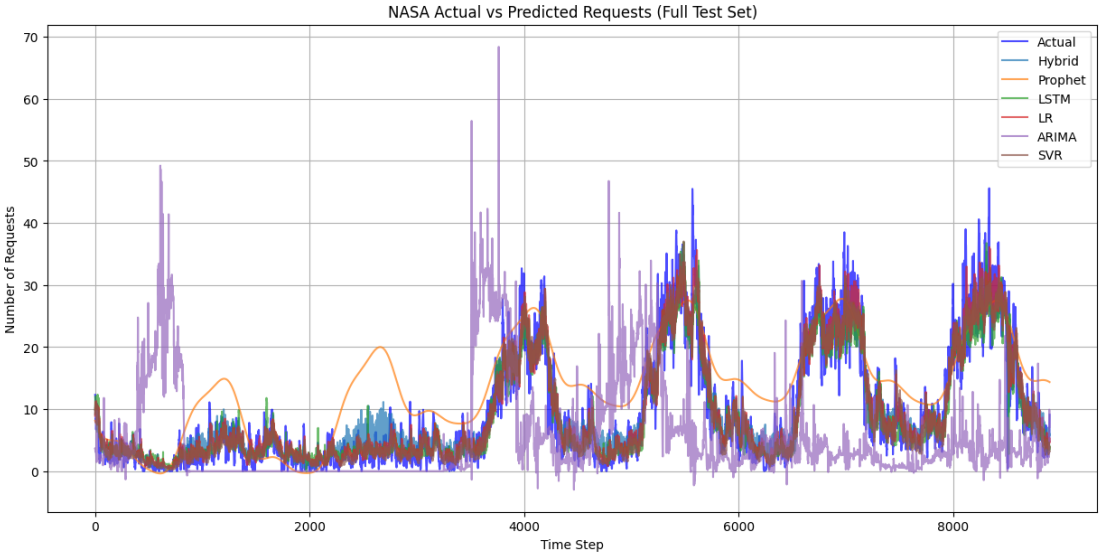
TABLE 3.3: Prediction Accuracy on NASA HTTP Logs (W = 60 min, H = 10 min)

Model	**MAE**	**RMSE**	**MAPE**	**R^2**
Only LR	2.512523	3.466961	2.56E+06	0.855353
SVR	2.517843	3.539463	2.23E+06	0.849240
Only LSTM	2.556482	3.545775	3.65E+06	0.848702
TCN	2.725441	3.704726	3.85E+06	0.834833
Only Prophet	4.499269	5.891381	4.74E+06	0.582318
ARIMA	8.119328	10.05128	1.55E+07	−0.21578
Hybrid (Proposed)	**2.492363**	**3.415395**	2.93E+06	**0.859265**

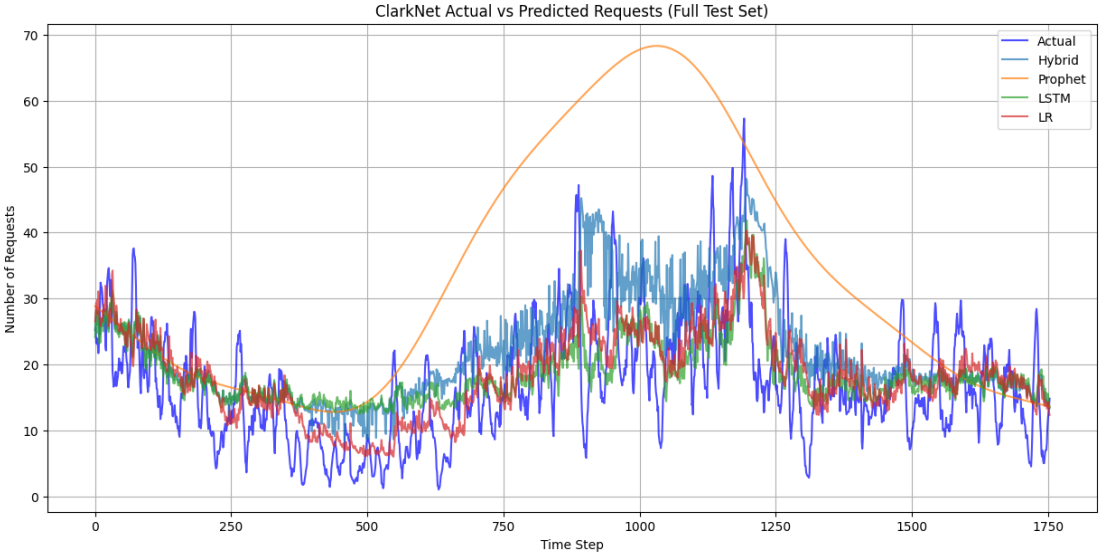
TABLE 3.4: Prediction Accuracy on ClarkNet WWW (W = 60 min, H = 10 min)

Model	**MAE**	**RMSE**	**MAPE**	**R^2**
Only LR	5.811724	7.355730	52.654869	0.317038
Only LSTM	5.995649	7.530014	57.850128	0.284291
TCN	5.928095	7.547170	57.118645	0.281026
SVR	6.202307	7.756597	62.893710	0.240570
ARIMA	9.965972	11.716713	123.192620	−0.732833
Only Prophet	17.74555	22.873599	146.507280	5.604109
Hybrid (Proposed)	**5.806198**	**7.338094**	52.901305	0.307553

Figure 3.12 compares the actual and predicted workloads over the complete time series for a window size of 60 minutes and a prediction horizon of 10 minutes. Figure 3.13 provides a zoomed view of the prediction results to highlight the model's ability to closely follow short-term workload variations. Figure 3.14 presents representative prediction intervals for the NASA and ClarkNet datasets, demonstrating the consistency of the proposed model under different workload patterns. Finally, Figure 3.15 compares the cumulative distribution functions (CDF) of the actual workload, the proposed hybrid ensemble, and the individual prediction models, illustrating the superior distributional agreement achieved by the hybrid ensemble across both datasets.

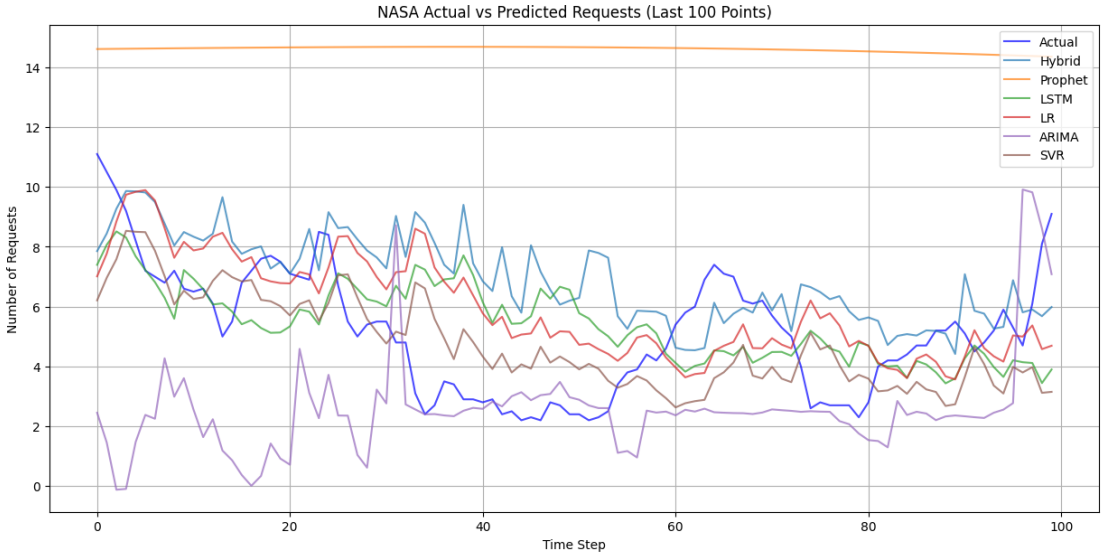


(A) NASA: actual vs. predicted workload (full trace)

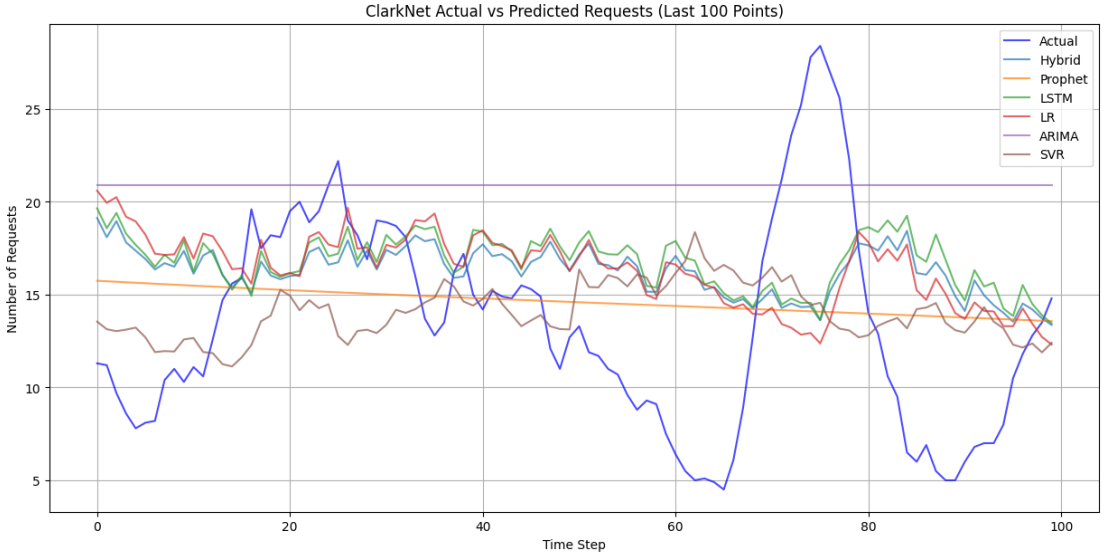


(B) ClarkNet: actual vs. predicted workload (full trace)

FIGURE 3.12: Actual vs. predicted workload over the complete time series (Hybrid ensemble, $W = 60$ min, $H = 10$ min).

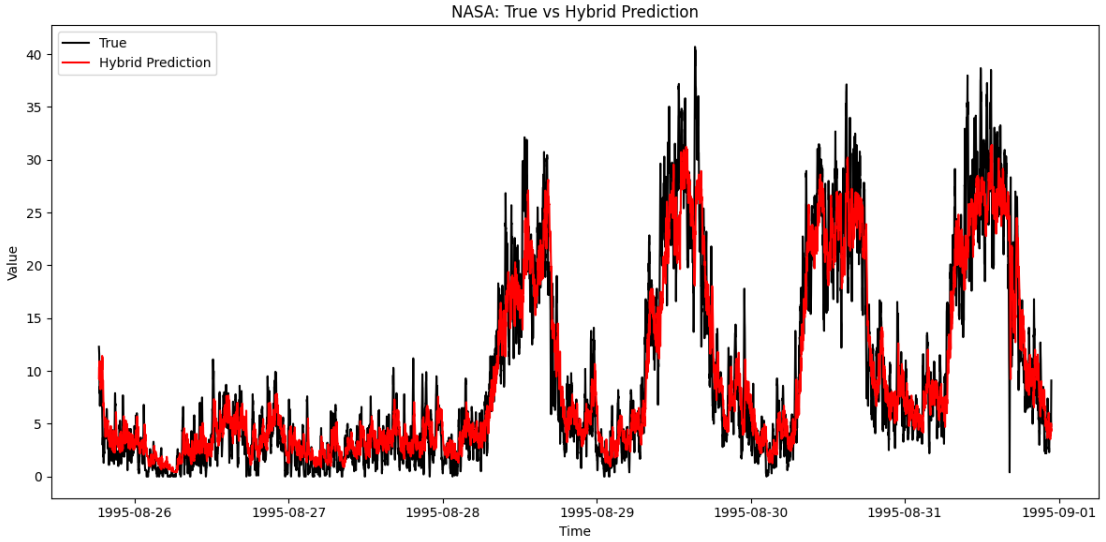


(A) NASA: zoomed view (100 points)

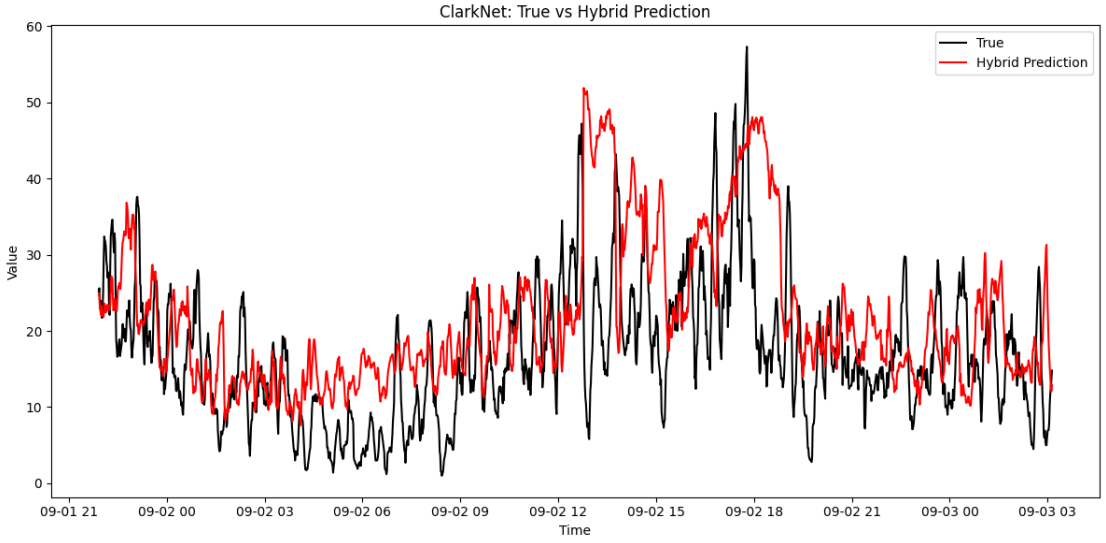


(B) ClarkNet: zoomed view (100 points)

FIGURE 3.13: Zoomed comparison of actual vs. predicted workload (Hybrid ensemble, $W = 60$ min, $H = 10$ min).



(A) NASA: sample interval illustration



(B) ClarkNet: sample interval illustration

FIGURE 3.14: Representative prediction intervals for NASA and ClarkNet using the hybrid ensemble.

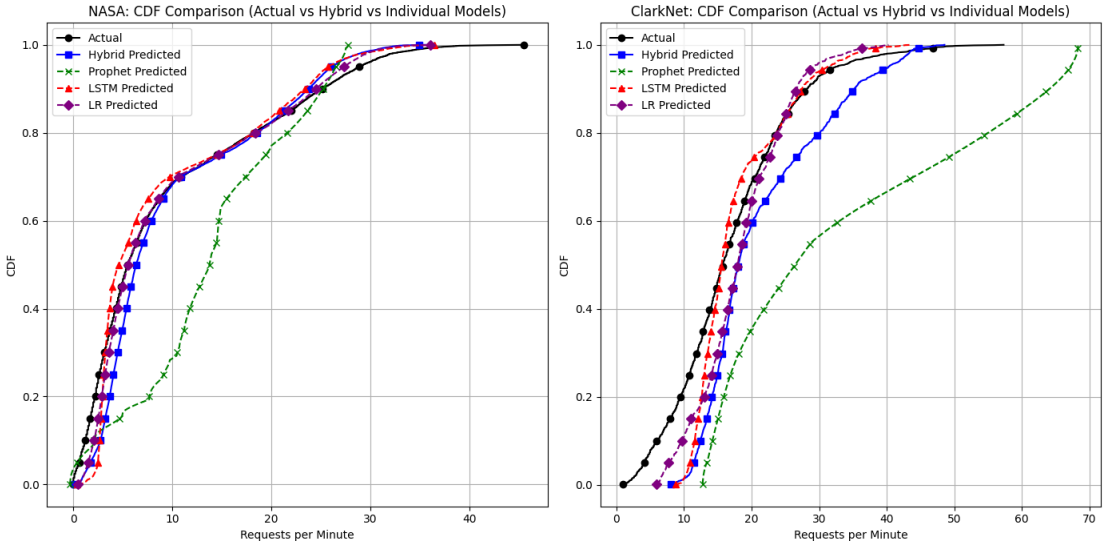


FIGURE 3.15: CDF Comparison (Actual vs Hybrid Vs Individual Models for NASA and ClarkNet)

On the NASA dataset, the hybrid ensemble achieves the lowest MAE (2.492) and RMSE (3.415) and the highest R^2 (0.859) among all evaluated models. ARIMA performs worst, with an RMSE of 10.051 and a negative R^2, confirming that linear statistical models are inadequate for the non-stationary NASA workload. Standalone Prophet also performs poorly (RMSE = 5.891) despite the dataset's periodicity because the 10-minute horizon is shorter than the dominant daily cycle that Prophet is optimised for. The hybrid ensemble improves on standalone LR, the next best model, by 1.0% in MAE and 1.2% in RMSE, confirming that the ensemble fusion provides a consistent, if modest, improvement over the best individual component.

On the ClarkNet dataset, the hybrid ensemble achieves the lowest MAE (5.806) and RMSE (7.338). The absolute accuracy values are higher than at NASA, reflecting the greater irregularity of the ClarkNet workload. ARIMA and standalone Prophet perform particularly poorly, with RMSE values of 11.717 and 22.874, respectively, confirming that models optimized for stationary or strongly seasonal patterns are unsuitable for irregular workloads. The hybrid ensemble's improvement over standalone LR (the next best model) is marginal on ClarkNet (0.09% in MAE), indicating that the non-linear components of the ClarkNet workload are not fully captured by the current ensemble. This observation motivates the future work direction of incorporating other models like transformer-based models with longer attention windows.

The hybrid's strength lies in its adaptability, driven by clustering and dynamic weighting, reducing RMSE significantly on both datasets. This aligns with hybrid trends in [109], who emphasized multi-model integration. However, the ClarkNet underperformance suggests a limitation in handling extreme nonlinearity, possibly due to LSTM's initialization sensitivity [5] or clustering over-segmentation. The NASA MAPE indicates scale issues, potentially addressable by adjusting loss functions.

3.11.2 Sensitivity to Window Size and Prediction Horizon

Tables 3.5 and 3.6 present the RMSE and MAE of the hybrid ensemble across five window sizes ($W \in \{30, 60, 90, 120, 150\}$ minutes) and five horizons ($H \in \{10, 15, 20, 25, 30\}$

minutes).

TABLE 3.5: RMSE Sensitivity to Window Size and Horizon (Hybrid Ensemble Model)

Dataset	W **(min)**	**Prediction horizon** H **(min)**				
		10	**15**	**20**	**25**	**30**
NASA	30	3.800	3.876	3.954	4.033	4.113
	60	3.415	3.484	3.553	3.624	3.697
	90	3.245	3.310	3.376	3.443	3.512
	120	3.082	3.143	3.206	3.270	3.335
	150	2.928	2.987	3.047	3.108	3.170
ClarkNet	30	8.200	8.364	8.531	8.702	8.876
	60	7.338	7.485	7.635	7.787	7.943
	90	6.971	7.111	7.253	7.398	7.546
	120	6.623	6.755	6.890	7.029	7.170
	150	6.291	6.417	6.546	6.677	6.810

Several trends are evident. First, RMSE decreases monotonically as W increases for both datasets, confirming that longer historical context improves prediction accuracy. Second, RMSE increases monotonically as H increases, confirming that longer horizons are inherently harder to predict. Third, the sensitivity to H is approximately linear: each additional 5-minute increment in horizon increases RMSE by approximately 2% on NASA and 2% on ClarkNet. This regularity suggests that the ensemble's error growth with horizon is well-behaved and predictable.

It was observed that increasing the window size improves prediction performance only up to a certain point. Beyond this stabilization point, further increases provide little or no improvement in prediction accuracy and may even degrade performance due to the inclusion of redundant historical information. This indicates that there is no universally optimal window size. Instead, the window size (w) should be treated as a hyperparameter whose optimal value depends on the characteristics of the dataset and the prediction model. The results suggest that while larger window sizes initially enhance model performance, the benefits diminish after a certain threshold. Therefore, rather than selecting a fixed window size for all applications, the objective should be to identify the stabilization point that best suits the specific dataset and forecasting model.

TABLE 3.6: MAE Sensitivity to Window Size and Horizon (Hybrid Ensemble Model)

Dataset	W **(min)**	**Prediction horizon** H **(min)**				
		10	**15**	**20**	**25**	**30**
NASA	30	2.700	2.757	2.816	2.876	2.938
	60	2.492	2.545	2.598	2.653	2.710
	90	2.368	2.417	2.467	2.519	2.572
	120	2.249	2.295	2.342	2.391	2.441
	150	2.137	2.181	2.225	2.270	2.317
ClarkNet	30	6.100	6.225	6.353	6.484	6.619
	60	5.806	5.921	6.039	6.160	6.283
	90	5.516	5.626	5.739	5.854	5.971
	120	5.240	5.345	5.452	5.561	5.672
	150	4.978	5.078	5.179	5.283	5.388

3.12 Summary

This chapter presented the clustering-enhanced hybrid ensemble workload prediction model, which constitutes the Analyse phase of the MAPE-K framework. The model extracts three workload features (variance, periodicity, and rate of change) from a 60-minute sliding window, applies K-means clustering to identify the current workload regime among three clusters (seasonal, non-linear, and steady-state), and assigns dynamic ensemble weights to three base models (Prophet, LSTM, Ridge LR) based on cluster proximity and recent prediction error. On the NASA dataset at $W = 60$ min and $H = 10$ min, the model achieves MAE = 2.492, RMSE = 3.415, and $R^2 = 0.859$, outperforming SVR, Bi-LSTM, TCN, standalone Prophet, and ARIMA. On ClarkNet, it achieves MAE = 5.806 and RMSE = 7.338. The sensitivity analysis confirms that accuracy improves with longer windows and shorter horizons.

Chapter 4 describes how the ensemble prediction output $\hat{A}(t+H)$ is translated into scaling decisions through the proactive auto-scaling decision model.

Chapter 4

Proactive auto-scaling Model

4.1 Overview

Workload prediction, however accurate, is only one half of a proactive auto-scaling system. The other half is the resource estimation model which is the mechanism that translates a predicted workload value into a concrete replica count and issues the scaling command at the right time. A naive resource estimation model that directly converts each prediction to a replica count will oscillate, over-provision during noisy predictions, and fail to respect the practical constraints of a live cluster. This chapter presents the proactive auto-scaling model, which is the Plan phase of the MAPE-K control loop.

The Proactive auto-scaling model incorporates stability controls that address the most common failure modes of prediction-driven autoscalers: oscillation, over-provisioning, under-provisioning during latency spikes, and scale-in thrashing. Together with the hybrid ensemble workload prediction model described in Chapter 3, the proactive auto-scaling model forms the complete proactive auto-scaling algorithm that is evaluated in Chapters 5 and 6.

The remainder of this chapter is organized as follows. Section 4.2 explains resource estimation in the Plan phase and describes how capacity profiling is used to estimate the maximum sustainable throughput per replica. Section 4.3 presents the proposed proactive auto-scaling algorithm, including prediction smoothing, workload blending, replica estimation, bounds enforcement, cooldown control, downscale confirmation, and latency-based protection. Section 4.4 describes the reactive CPU-based auto-scaling baseline used for comparison. Section 4.5 compares the proactive and reactive models in terms of scaling trigger, prediction horizon, control signal, and stability mechanisms. Section 4.6 discusses the sensitivity of key parameters. Finally, Section 4.7 summarizes the chapter

and links the proposed auto-scaling model to the implementation and evaluation presented in the following chapters.

4.2 Resource Estimation in the Plan Phase

Existing auto-scaling studies classify resource-estimation methods into six main categories: rule-based approaches, fuzzy inference, capacity profiling, analytical modeling, machine-learning approaches and hybrid approaches [6].

Rule-based approaches use predefined conditions and actions, such as adding replicas when CPU utilization exceeds an upper threshold or removing replicas when utilization falls below a lower threshold. These methods are simple and widely used, but their performance is heavily dependent on threshold values and scaling step sizes; inappropriate values may lead to over-provisioning or delayed scaling after performance degradation has already started [110].

Fuzzy-inference approaches extend rule-based planning by replacing sharp numerical thresholds with linguistic terms such as low, medium, and high. They are more flexible than strict threshold rules, but still require carefully designed membership functions and rule sets [111].

Analytical modeling estimates resource requirements using mathematical relationships among workload arrival rate, service rate, utilization, response time, and number of servers. Queueing theory is commonly used in this category, where an application may be represented as an $M/M/1$, $M/M/c$, or queueing-network model depending on the service architecture. Here, M denotes Markovian arrival and service processes, while 1 or c represents the number of servers in the system. These models can estimate the number of servers required for a target response time or utilization level but require parameters such as the service-time distribution and the arrival process, which are difficult to measure accurately in real deployments. Salah et al. [112] proposed an analytical model for estimating resources of elastic cloud services, showing how mathematical modeling can support resource planning.

Machine-learning-based approaches learn the relationship between workload, resource allocation, and application performance from historical or runtime data. Regression, feedback-control, and adaptive learning models can estimate resource demand or predict performance under different allocations. These methods are more adaptive than fixed rules, but they may require sufficient training data and stabilization time before producing reliable decisions. Padala et al. [113] demonstrated automated control of multiple virtualized resources, while Gambi et al. [114] proposed Kriging-based self-adaptive cloud controllers.

Hybrid approaches combine two or more resource-estimation methods to reduce the limitations of individual techniques. For example, a system may combine analytical modeling with machine learning, or profiling with runtime monitoring, to improve estimation accuracy and adaptability. [115] used hybrid reinforcement learning for autonomic resource allocation, showing how learning-based and model-based ideas can be combined for adaptive resource management.

While hybrid approaches (e.g., analytical + ML) offer adaptability [115], the planner in this thesis adopts a capacity-profiling approach for its simplicity, stability, and suitability for stable microservice codebases. This choice aligns with prior works on offline profiling [116, 117] while integrating seamlessly with the RPS-based ensemble predictions.

4.2.1 Capacity Profiling for Estimating $T_{\max}$

Capacity Profiling is a process to test the saturation point of resources when running the specific application using a synthetic or recorded real workload. Application profiling is the simplest way to accurately acquire the knowledge of how many resources are just enough to handle the given workload intensity concurrently. Offline profiling has been used by [118, 116, 117], while online profiling and fast resource estimation have been explored by [119, 120]. Capacity profiling is particularly effective in environments where microservices or application code remain relatively stable and are not updated frequently, ensuring that the derived capacity estimates remain valid over time. Here, $T_{\max}$ represents

the maximum request rate that one replica can handle while still satisfying the p95 latency SLA.

Here, the per-replica throughput capacity $T_{\max}$ is obtained via an offline capacity-profiling experiment as depicted in Algorithm 2. The service is first scaled to a fixed replica count R, and then a Locust workload is increased in steps until the observed p95 latency exceeds the SLA threshold. The maximum sustainable request rate per replica is then computed as $T_{\max} = A_{\text{rps}}/R$.

Algorithm 2 Capacity Profiling for Estimating $T_{\max}$

Require: Service S, SLA latency threshold SLA_{p95}, initial users U_{start}, step size ΔU

Ensure: Maximum sustainable request rate per replica $T_{\max}$

Scale service to fixed replicas R
$U \leftarrow U_{\text{start}}$
while $U \leq U_{\max}$ **do**
    Generate workload using Locust with U users
    Run experiment for duration T
    Collect metrics from Prometheus: $A_{\text{rps}} \leftarrow$ observed request rate, $L_{p95} \leftarrow$ 95th percentile latency
    if $L_{p95} \leq \text{SLA}_{p95}$ **then**
        Record A_{rps} as sustainable throughput
        $U \leftarrow U + \Delta U$
    else
        break
    end if
end while
$T_{\max} \leftarrow \dfrac{A_{\text{rps}}}{R}$
return $T_{\max}$

The resulting $T_{\max}$ value is stored and periodically can be refreshed to account for minor code or configuration changes. In the experimental setup (Chapter 5), $T_{\max}$ was determined using Locust-generated workloads on a representative microservice.

4.3 Proactive auto-scaling Algorithm

While workload prediction provides the foresight required for proactive scaling, several practical challenges can hinder the effectiveness of prediction-driven auto-scaling systems. These threats, if unaddressed, often lead to instability, poor resource efficiency, or service level objective (SLO) violations in production environments. The proposed proactive auto-scaling algorithm systematically identifies and mitigates the following key threats:

- **Prediction Noise and Oscillation**: Raw ensemble predictions can fluctuate significantly during workload transitions, causing unnecessary scale-out and scale-in actions (scaling oscillation or thrashing).
- **Over-provisioning**: Overly optimistic predictions or failure to account for prediction errors can result in excessive resource allocation, increasing operational costs.
- **Under-provisioning during Spikes**: Sudden demand surges that exceed the prediction horizon or model accuracy can lead to latency spikes and SLA violations.
- **Premature Scale-in**: Aggressive reduction in replica count during temporary dips in demand can cause immediate SLA breaches if workload rebounds.
- **Cluster Resource Constraints**: Unbounded scaling decisions may exhaust available cluster resources or violate operational limits.
- **Delayed Reaction to Unpredicted Events**: Purely predictive approaches lack fallback mechanisms when predictions deviate significantly from reality.

To overcome these challenges, the proposed proactive auto-scaling algorithm incorporates multiple stability controls within the Plan phase of the MAPE-K loop. These controls transform the raw hybrid ensemble prediction $\hat{A}(t+H)$ into a reliable and stable scaling decision $R^*(t)$. The algorithm consists of seven integrated steps. The complete algorithm is presented in Algorithm 3 at the end of this section. Each component is described in

detail in the following subsections. Together, these mechanisms ensure robust, SLA-aware auto-scaling when combined with the hybrid ensemble predictor from Chapter 3.

4.3.1 Step 1: Prediction Smoothing

Raw ensemble predictions can be noisy, particularly during workload transitions when the cluster assignment changes. Applying a scaling command based on a single noisy prediction can cause unnecessary scale-out followed immediately by scale-in. To mitigate this, an exponential moving average (EMA) is applied to the raw prediction:

$$\hat{A}_s(t) = \lambda \hat{A}(t+H) + (1-\lambda)\hat{A}_s(t-1) \tag{4.1}$$

where $\hat{A}_s(t)$ is the smoothed prediction, $\hat{A}(t+H)$ is the raw ensemble prediction, $\hat{A}_s(t-1)$ is the previous smoothed value, and $\lambda \in (0,1)$ is the smoothing factor. A larger λ makes the smoothed prediction more responsive to the current raw prediction; a smaller λ makes it more conservative. The value of λ is set empirically based on validation set performance.

4.3.2 Step 2: Effective Workload Blending

Even after smoothing, relying entirely on the predicted workload can lead to over-provisioning when predictions are consistently higher than actual demand. Blending the smoothed prediction with the current observed workload provides a more conservative estimate:

$$A_{\text{eff}}(t) = \beta \hat{A}_s(t) + (1-\beta)A(t) \tag{4.2}$$

where $A_{\text{eff}}(t)$ is the effective workload used for replica calculation, $A(t)$ is the current observed workload, and $\beta \in (0,1)$ is the blending factor. When $\beta = 1$, the system is fully proactive; when $\beta = 0$, it is fully reactive. The blending factor is tuned to balance proactive anticipation against the risk of over-provisioning.

4.3.3 Step 3: Replica Count Estimation with Safety Margin

The target replica count is estimated from the effective workload using the measured per-replica throughput capacity $T_{\max}$ and a safety margin factor α, where $0 < \alpha \leq 1$. The required number of replicas is computed as:

$$R^*(t) = \left\lceil \frac{A_{\text{eff}}(t)}{\alpha \cdot T_{\max}} \right\rceil \tag{5.3}$$

where $A_{\text{eff}}(t)$ denotes the effective workload at time t, $T_{\max}$ denotes the measured maximum throughput capacity of one replica, and α denotes the usable-capacity factor considered during planning. The factor α derates the maximum capacity of each replica. Therefore, instead of assuming that every replica can continuously operate at its full measured throughput, the autoscaler plans replicas using only a fraction α of $T_{\max}$.

When $\alpha = 1$, the autoscaler assumes that each replica can be used up to its full measured capacity. In this case, the estimated replica count corresponds to the strict minimum required to serve the effective workload. When $\alpha < 1$, the usable capacity of each replica is reduced in the calculation. This increases the estimated number of replicas and provides additional headroom for prediction errors, short-term workload bursts, runtime variability, and measurement noise.

For example, if $\alpha = 0.9$, each replica is planned to handle only 90% of its measured maximum throughput. Hence, the estimated replica requirement becomes approximately $1/0.9 = 1.111$ times the strict full-capacity requirement, which corresponds to about 11.1% additional capacity headroom before applying integer rounding. Similarly, if $\alpha = 0.85$, each replica is planned to handle only 85% of its measured maximum throughput, giving approximately $1/0.85 = 1.176$, or 17.6%, additional capacity headroom before rounding.

4.3.4 Step 4: Replica Bounds Enforcement

The target replica count is clamped to the feasible range:

$$R^*(t) \leftarrow \max(R_{\min}, \min(R^*(t), R_{\max})) \tag{4.3}$$

where $R_{\min} = 1$ and $R_{\max} = 12$. The minimum bound ensures that the service remains available even during very low demand periods; the maximum bound prevents the cluster from being exhausted by a runaway scaling event.

4.3.5 Step 5: Cooldown Enforcement

To prevent rapid oscillation between scale-out and scale-in, a cooldown period $T_c = 60$ seconds is enforced between consecutive scaling actions:

$$\text{scale if } (t - t_{\text{last}}) \geq T_c \tag{4.4}$$

where t_{last} is the timestamp of the most recent scaling action. If the cooldown has not elapsed, the scaling command is suppressed regardless of the target replica count.

4.3.6 Step 6: Downscale Confirmation

Scale-in actions are inherently riskier than scale-out actions because removing replicas reduces capacity and may cause SLA violations if demand rebounds. To reduce false scale-in events, a confirmation counter $v_a(t)$ is maintained:

$$v_a(t) = \begin{cases} v_a(t-1) + 1 & \text{if } R^*(t) < R(t) \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

A scale-in action is executed only when $v_a(t) \geq k_a$, where k_a is the required number of consecutive confirmations. This mechanism prevents scale-in from occurring after a

single low-demand observation, requiring the system to observe sustained low demand before reducing capacity.

4.3.7 Step 7: Latency Guard

The final stability control is a latency-aware safety mechanism that prevents unsafe scale-in when the application is already experiencing an SLA violation. Although the predicted workload may indicate that fewer replicas are required, reducing the available capacity while the observed response latency is high may further degrade application performance. Therefore, the current replica count is retained whenever the predicted target replica count is lower than the current replica count and the observed 95th-percentile latency exceeds the predefined SLA threshold.

The latency-aware scale-in guard is defined as:

$$\text{if } R^*(t) < R(t) \ \wedge \ L_{95}(t) > L_{\text{SLA}}, \quad R^*(t) \leftarrow R(t), \tag{4.6}$$

where $R^*(t)$ denotes the target replica count determined by the proactive auto-scaling model, $R(t)$ represents the currently active replica count, $L_{95}(t)$ is the observed 95th-percentile response latency at time t, and L_{SLA} is the predefined SLA latency threshold. In the present experimental framework, L_{SLA} is set to 200 ms.

This mechanism ensures that the auto-scaler does not remove active replicas during periods of degraded application performance. For example, if the prediction-based planning model recommends reducing the number of replicas from five to three while the observed $p95$ latency is above the SLA threshold, the scale-in action is suppressed and the replica count remains at five. Thus, the latency guard provides protection against premature resource reduction when the predicted workload and the observed application-level performance indicate conflicting system conditions.

The primary scaling decision remains prediction-driven; however, the use of observed latency introduces a reactive feedback constraint into the decision process. Consequently, the mechanism improves the robustness of the proactive auto-scaling framework against

prediction errors and temporary workload fluctuations without independently initiating an additional scale-out action.

4.3.8 Proactive Algorithm

Algorithm 3 presents the complete proactive auto-scaling algorithm.

Algorithm 3 Proactive RPS-Driven auto-scaling Algorithm

Require: Current replicas $R(t)$; observed RPS $A(t)$; predicted RPS $\hat{A}(t+H)$; previous smoothed prediction $\hat{A}_s(t-1)$; p95 latency $L_{95}(t)$; per-replica capacity $T_{\max}$; safety factor α; prediction smoothing factor λ; blending factor β; minimum replicas $R_{\min}$; maximum replicas $R_{\max}$; cooldown interval τ_c; last scale time t_{last}; latency SLA threshold L_{SLA}; downscale confirmation counter $v_d(t-1)$; required confirmations k_d

Ensure: Target replicas $R^*(t)$

1: **if** $t - t_{\text{last}} < \tau_c$ **then**
2: **return** $R(t)$ ▷ Skip scaling during cooldown
3: **end if**
4: $\hat{A}_s(t) \leftarrow \lambda\hat{A}(t+H) + (1-\lambda)\hat{A}_s(t-1)$ ▷ Smooth prediction
5: $A_{\text{eff}}(t) \leftarrow \beta\hat{A}_s(t) + (1-\beta)A(t)$ ▷ Blend predicted and observed workload
6: $R^*(t) \leftarrow \left\lceil \dfrac{A_{\text{eff}}(t)}{\alpha T_{\max}} \right\rceil$ ▷ Compute required replicas
7: $R^*(t) \leftarrow \min(\max(R^*(t), R_{\min}), R_{\max})$ ▷ Apply replica bounds
8: **if** $R^*(t) < R(t)$ **and** $v_d(t-1) < k_d$ **then**
9: $R^*(t) \leftarrow R(t)$ ▷ Block premature scale-in
10: **end if**
11: **if** $R^*(t) < R(t)$ **and** $L_{95}(t) > L_{\text{SLA}}$ **then**
12: $R^*(t) \leftarrow R(t)$ ▷ Block scale-in during SLA violation
13: **end if**
14: **return** $R^*(t)$

In Algorithm 3, Lines 1–3 check the cooldown condition and return the current replica count if scaling is not allowed yet. Line 4 smooths the predicted RPS using exponential moving average to reduce prediction noise. Line 5 combines the smoothed predicted

workload with the current observed workload to obtain the effective workload. Line 6 calculates the required replica count using the effective workload, per-replica capacity $T_{\max}$, and safety factor α. Line 7 restricts the computed replica count within the minimum and maximum replica bounds. Lines 8–10 block premature scale-in if the required number of downscale confirmations has not been reached. Lines 11–13 prevent scale-in when the p95 latency is higher than the SLA threshold. Finally, Line 14 returns the target replica count $R^*(t)$.

4.4 Reactive auto-scaling Baseline

The reactive autoscaler computes the target number of replicas from the ratio between current CPU utilization and target CPU utilization. If the current CPU utilization exceeds the target, the replica count increases proportionally; if it is below the target, the replica count decreases proportionally. Reactive scaling is inspired by Kubernetes Horizontal Pod Autoscaler (HPA) [121].

Algorithm 4 presents the reactive auto-scaling algorithm.

Algorithm 4 Reactive CPU-Based auto-scaling

Require: Current replicas $R(t)$; current CPU utilization $U(t)$; target CPU utilization U_{target}; minimum replicas $R_{\min}$; maximum replicas $R_{\max}$; cooldown interval τ_c; last scaling time t_{last}; small constant ϵ

Ensure: Target replicas $R^*(t)$

1: **if** $t - t_{\text{last}} < \tau_c$ **then**
2: **return** $R(t)$ ▷ Skip scaling during cooldown
3: **end if**
4: $R^*(t) \leftarrow \left\lceil R(t) \times \dfrac{U(t)}{\max(U_{\text{target}}, \epsilon)} \right\rceil$ ▷ Compute target replicas from CPU utilization ratio
5: $R^*(t) \leftarrow \min(\max(R^*(t), R_{\min}), R_{\max})$ ▷ Apply replica bounds
6: **return** $R^*(t)$

In Algorithm 4, Lines 1–3 check the cooldown condition and return the current replica count if the cooldown period has not elapsed. Line 4 computes the target replica count using the ratio between current CPU utilization (U(t)) and target CPU utilization U_{target}), following an HPA-like scaling rule. Line 5 restricts the calculated replica count within the minimum and maximum replica limits. Finally, Line 6 returns the target replica count $R^*(t)$.

4.5 Comparison of Proactive and Reactive auto-scaling Models

Table 4.1 summarises the key differences between the proactive and reactive auto-scaling models.

Table 4.1: Comparison of Proactive and Reactive auto-scaling auto-scaling Models

Property	**Proactive (Proposed)**	**Reactive (Baseline)**
Scaling trigger	Predicted workload $\hat{A}(t+H)$	Observed CPU utilisation $U(t)$
Scaling signal	RPS (request rate)	CPU utilisation (%)
Startup lag compensation	Yes (10-minute lead time)	No
Prediction smoothing	EMA with factor λ	N/A
Effective workload blending	Yes (β)	N/A
Safety margin	Yes (α)	N/A
Downscale confirmation	Yes (k_a cycles)	No
Latency guard	Yes (p95 > 200 ms)	Emergency CPU override
Cooldown period	60 seconds	60 seconds
Replica bounds	$R_{\min} = 1$, $R_{\max} = 12$	$R_{\min} = 1$, $R_{\max} = 12$

The key structural difference is the prediction horizon: the proactive system issues scaling commands h minutes before the demand arrives, giving the Docker Swarm manager sufficient time to schedule and start new container replicas before the workload peaks. The reactive system can only respond after the demand has already arrived, meaning that new replicas become operational only after the SLA has been violated. The latency guard in the proactive system provides a reactive safety net for the cases where the prediction is insufficient, making the overall system a hybrid proactive-reactive design.

4.6 Parameter Sensitivity

The proactive auto-scaling model has three principal tuning parameters: the smoothing factor λ, the blending factor β, and the safety margin α. The interaction between these parameters determines the trade-off between SLA compliance and resource overhead.

A higher λ makes the smoothed prediction more responsive to the latest raw prediction, which improves responsiveness but increases sensitivity to noise. A higher β gives more weight to the predicted workload relative to the current observed workload, which increases proactivity but also increases over-provisioning risk. A lower α increases the safety margin (more replicas than strictly needed), which reduces SLA violations but increases resource cost.

The parameters are tuned on the validation set by grid search over $\lambda \in \{0.3, 0.5, 0.7\}$, $\beta \in \{0.5, 0.7, 0.9\}$, and $\alpha \in \{0.85, 0.9, 1.0\}$, selecting the combination that minimises the SLA violation rate subject to a resource overhead constraint of no more than 10% additional replicas compared to the reactive baseline. The selected values are reported in Chapter 5 as part of the experimental configuration.

4.7 Summary

This chapter presented the proactive auto-scaling model, which translates the ensemble workload prediction into concrete scaling commands through stability controls: prediction smoothing, effective workload blending, safety margin, replica bounds, cooldown enforcement, downscale confirmation, and latency guard. The model is designed to balance proactive anticipation against the risks of over-provisioning and oscillation, and incorporates a reactive latency guard as a safety net for prediction errors. The reactive auto-scaling baseline, which uses CPU utilisation thresholds without prediction, was also described and compared against the proactive model. The complete proactive and reactive algorithms were presented in pseudocode form.

Chapter 5 describes the physical and software infrastructure on which both algorithms are deployed and evaluated.

Chapter 5

System Architecture and MAPE-K-Based Experimental Framework

5.1 Overview

This chapter describes the physical infrastructure, software stack, deployment configuration, workload generation mechanism, and monitoring pipeline that together form the experimental framework used to evaluate the proposed proactive auto-scaling system. The framework is designed to provide a controlled, reproducible environment that closely approximates real-world cloud conditions while allowing all relevant metrics to be collected with system-level consistency.

Section 5.3 describes the overall system architecture and the realization of the MAPE-K control loop. Section 5.4 details the three-node Docker Swarm cluster infrastructure deployed on Amazon EC2. Section 5.5 explains the deployed software stack, target CPU-intensive application, and the service-task-container relationship in Docker Swarm. Section 5.6 covers the monitoring and metrics collection pipeline using Prometheus and cAdvisor. Section 5.7 presents the workload generation mechanism using both real-world traces (NASA and ClarkNet) and synthetic patterns (increasing and periodic). Section 5.8 defines the four experimental configurations and the evaluation metrics. Section 5.9 summarises the proactive auto-scaler parameter settings, while Section 5.10 outlines the automated experiment execution workflow.

The framework implements the MAPE-K control loop on a live three-node Docker Swarm cluster hosted on Amazon EC2. All workload patterns are injected through a

trace-driven Locust load generator [122] that replays pre-processed versions of the NASA HTTP [103] and ClarkNet datasets [104], as well as two synthetic patterns (monotonically increasing and periodic). All performance metrics are collected exclusively through the Prometheus and cAdvisor tools, ensuring that the evaluation is independent of the load generation mechanism.

5.2 Proposed System Architecture

The architecture and workflow of the proposed proactive auto-scaling framework deployed on a Docker Swarm cluster is illustrated in Figure 5.1. The diagram shows the complete flow from workload submission and request routing to service discovery, request handling, metrics collection, workload prediction, scaling decision, scaling trigger, replica scheduling, and updated container deployment. Client and external user requests are routed through the ingress/load balancer and handled by container replicas running on worker nodes. Runtime metrics collected through Prometheus and cAdvisor are used by the auto-scaling policy to predict future workload demand, after which the autoscaler triggers scaling actions and the Swarm scheduler assigns updated replicas across worker nodes.

The proposed methodology consists of two major models: a **workload forecasting model** and a **proactive auto-scaling model**. The workload forecasting model predicts the future request rate using a clustering-based hybrid ensemble of Prophet, LSTM, and linear regression. The proactive auto-scaling model converts the predicted workload into a target number of container replicas before workload demand fully appears.

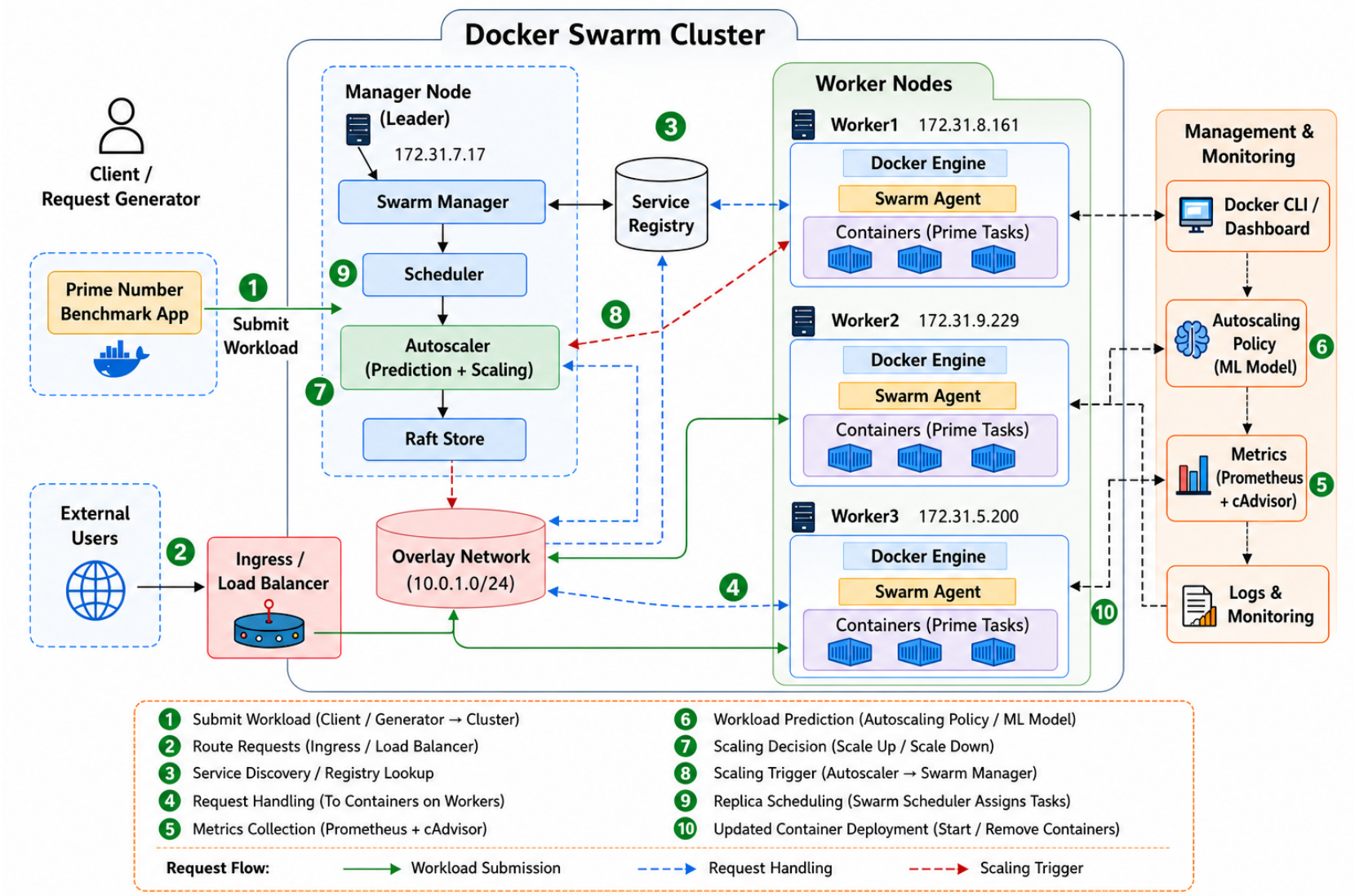


FIGURE 5.1: Docker Swarm-Based Auto-scaling Architecture

5.3 MAPE-K-Based Control Loop Realization

The MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) model is a standard architectural pattern for self-adaptive systems [123].

In the proposed framework, each phase is implemented as follows:

Monitor: Prometheus scrapes metrics from the application and cAdvisor every 10 seconds. The auto-scaler queries Prometheus every 10 seconds to retrieve the current request rate $A(t)$, the p95 latency $L_{95}(t)$, the average CPU utilisation per replica, and the active replica count $R(t)$.

Analyse: The clustering-enhanced hybrid ensemble model (Chapter 3) processes the most recent w window to produce the predicted workload $\hat{A}(t+H)$ at a h horizon.

Plan: The decision model (Chapter 4) applies stability controls to translate $\hat{A}(t+H)$ into a target replica count $R^*(t)$.

Execute: The auto-scaler issues a `docker service scale` command to the Docker Swarm manager, which schedules or removes container replicas on the worker nodes.

Knowledge: The knowledge base stores the system parameters: per-replica throughput capacity $T_{\max}$ (RPS), SLA threshold L_{SLA}, cooldown period T_c, replica bounds $R_{\min}$ and $R_{\max}$, and the historical workload window.

As shown in Figure 5.2, the framework is executed in two stages. In the first stage, historical workload traces from the NASA and ClarkNet web server logs, which are standard benchmark datasets, are used for model training. The workload traces are processed to extract request-rate time series, workload features are computed, workload patterns are identified using clustering, and the base forecasting models are trained. The trained models, cluster centroids, feature scalers, and model metadata are saved as artifacts.

In the second stage, the online auto-scaling loop is executed inside the containerized environment. Prometheus and cAdvisor collect runtime metrics such as observed RPS, CPU utilization, latency, and current replica count. The auto-scaler uses recent RPS history and the saved forecasting artifacts to infer future workload. The proactive auto-scaling model then computes the target replica count and Docker Swarm applies the scaling decision.

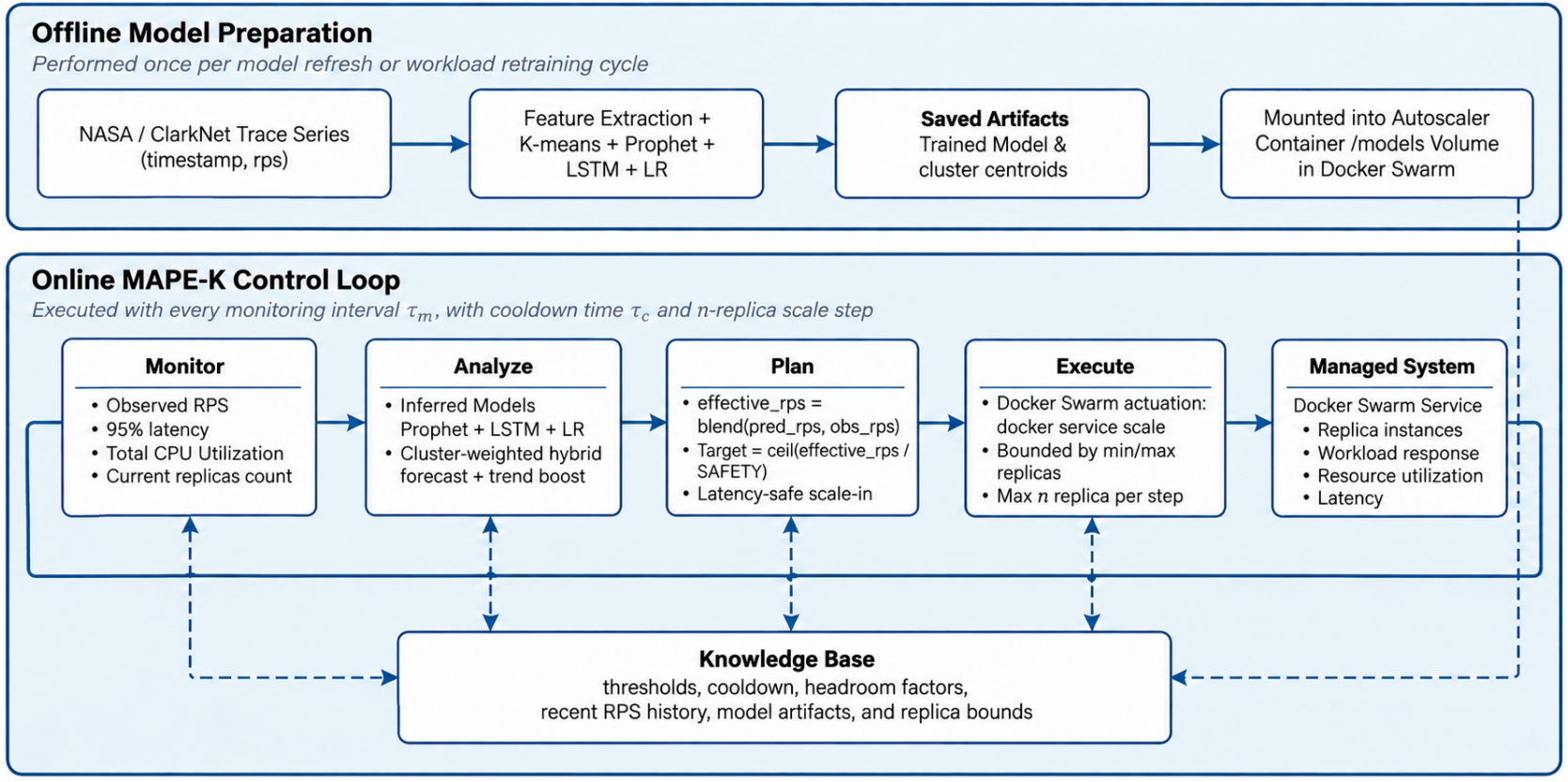


FIGURE 5.2: MAPE-K model for proactive auto-scaling mechanism

5.4 Experimental Cluster Infrastructure

This section describes the physical and virtual cluster infrastructure used to implement and evaluate the proposed proactive auto-scaling framework.

5.4.1 Experimental Cluster Infrastructure

To evaluate the proposed proactive auto-scaling system in a realistic setting, a three-node Docker Swarm cluster was deployed on Amazon EC2. This infrastructure provides a controlled environment that mimics production cloud conditions while enabling reproducible experiments and comprehensive metric collection.

The following subsections describe the physical resources, Swarm configuration, and deployed software stack.

5.4.2 Physical and Virtual Infrastructure

The experimental platform is deployed on three Amazon EC2 instances provisioned within the same Virtual Private Cloud (VPC) to ensure low-latency inter-node communication. All nodes run Amazon Linux 2. Table 5.1 summarizes the hardware and software configuration of each node.

TABLE 5.1: Cluster Node Configuration

Node	Role	CPU Model	vCPUs	RAM	Storage	OS	Docker
leader	Swarm Manager	Intel Xeon E5-2686 v4	2	4 GB	32 GB	Amazon Linux 2	25.0.6
worker1	Worker Node	Intel Xeon E5-2686 v4	2	4 GB	16 GB	Amazon Linux 2	20.10.7
worker2	Worker Node	Intel Xeon E5-2686 v4	2	4 GB	8 GB	Amazon Linux 2	20.10.7

The manager node is responsible for cluster management, scheduling decisions, and execution of scaling commands via the Docker Swarm service API. The worker nodes host the application containers and handle all incoming workload requests. The manager node also runs the autoscaler service and the Prometheus monitoring stack.

5.4.3 Docker Swarm Configuration

Docker Swarm [22] is used as the container orchestration platform. The cluster is initialized on the leader node and the worker nodes are joined using Swarm join tokens. An overlay network named `autoscaler_autoscale` connects all services across nodes. The network is configured with subnet `10.0.1.0/24`, gateway `10.0.1.1`, scope Swarm, and driver Overlay. Docker Swarm's ingress routing mesh relies on the Linux IPVS subsystem, which performs Layer-4 load balancing using a weighted round-robin scheduling algorithm to distribute incoming connections across active service replicas. Scaling operations are performed by updating the service replica count via the `docker service scale` command issued by the autoscaler.

```
root@leader:~/proactive-swarm-autoscale
[root@leader proactive-swarm-autoscale]# docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
iot9m83m88s64ep8l0x3tqmfh *   leader     Ready     Active         Leader           25.0.6
oyu29aii98rsit6b1e5at1qw6     worker1    Ready     Active         Reachable        20.10.7
nutd5gkkdtiizpk3hjx72ozgv     worker2    Ready     Active                          20.10.7
[root@leader proactive-swarm-autoscale]# docker network ls
NETWORK ID     NAME                   DRIVER    SCOPE
512mjtr647xh   autoscaler_autoscale   overlay   swarm
73464864f4ca   bridge                 bridge    local
a39d8613657e   docker_gwbridge        bridge    local
7fcf864372b2   host                   host      local
d7pohfzstpaf   ingress                overlay   swarm
156b8b4b893f   none                   null      local
[root@leader proactive-swarm-autoscale]# docker service ls
ID             NAME                MODE         REPLICAS   IMAGE                                                PORTS
oj31uq7kaad0   oneshot_autoscaler   replicated   1/1        naimishatrivedi/proactive-autoscaler:bounded-v3
upxquyfjif8o   oneshot_cadvisor     global       3/3        google/cadvisor:v0.25.0
kyxxuxydv06i   oneshot_locust       replicated   1/1        naimishatrivedi/prime-locust:latest                  *:8089-
>8089/tcp
vjp1qmec4r68   oneshot_prime_app    replicated   1/1        naimishatrivedi/prime-app:latest2                    *:8080-
>8000/tcp
3fhmzzvu7lk6   oneshot_prometheus   replicated   1/1        prom/prometheus:v2.12.0                              *:9090-
>9090/tcp
j80f55rkay3o   oneshot_registry     replicated   1/1        registry:2                                           *:5000-
>5000/tcp
```

FIGURE 5.3: Docker Swarm Configuration

5.5 Software Stack and Service Deployment

All components of the experimental framework are containerized and deployed as Docker Swarm services on the cluster. This ensures consistency, reproducibility, and easy management of the monitoring, application, load generation, and auto-scaling services. The following subsections describe the deployed services and the target application.

5.5.1 Deployed Services

All components are deployed as Docker services within the Swarm cluster and attached to the `autoscaler_autoscale` overlay network. The complete experimental environment was deployed using a Docker Stack configuration file named `one_shot_stack.yml`. This stack file defines the overlay network, local registry, cAdvisor, Prometheus, Prime application service, proactive autoscaler, and Locust workload generator. It also specifies service placement constraints, resource limits, Prometheus configuration, autoscaler environment variables, replica bounds, cooldown period, prediction horizon, per-replica capacity parameters, and SLA thresholds. Thus, the stack file serves as the reproducible deployment specification for the experimental framework. Table 5.2 and Figure 5.3 list the deployed services.

TABLE 5.2: Software Stack and Container Images

Component	Docker Image	Version	Purpose
Application Service	prime-app-metrics	latest	CPU-intensive prime computation
Autoscaler	naimishatrivedi/proactive-autoscaler	realml2-fixed	Proactive scaling controller
Prometheus	prom/prometheus	v2.12.0	Metrics collection and storage
cAdvisor	google/cadvisor	v0.25.0	Container-level resource monitoring
Load Generator	naimishatrivedi/prime-locust	2.24.1	Trace-driven workload generation
Registry	registry	v2	Local image registry

5.5.2 Target Application

The target application is a CPU-intensive microservice named `oneshot_prime_app`, implemented using Flask 3.0.2 and served by Gunicorn 21.2.0 on Python 3.11. The application exposes a single HTTP endpoint `GET /prime?n=50000`, which computes all prime numbers up to $n = 50{,}000$ using a trial-division algorithm. The computational complexity is $O(n\sqrt{n})$, producing approximately 400 ms of CPU time per request on the experimental hardware. This makes the application representative of a CPU-bound microservice workload, where CPU utilisation scales linearly with request rate.

The application is stateless, allowing horizontal scaling without shared state or coordination between replicas. It is initially deployed with a single replica and is dynamically scaled between $R_{\min} = 1$ and $R_{\max} = 12$ replicas by the autoscaler. The per-replica

throughput capacity is $T_{\max}$ = 10.4 RPS, determined empirically by measuring the maximum sustainable request rate of a single replica under a sustained load test.

The application exports performance metrics via the Prometheus client library, including the HTTP request rate, request latency histogram (used to compute p95), and endpoint-level request counts. These metrics are scraped by Prometheus at 15-second intervals.

5.5.3 Docker Swarm Service–Task–Container Relationship

Docker Swarm represents a replicated application using the hierarchy of service, task, and container. A *service* defines the desired state of an application, including the container image, network, exposed ports, resource limits, and desired number of replicas. A *task* is the scheduling unit created by the Swarm manager to realize the desired state of the service. Each running task corresponds to one container instance of the application on a worker node.

In the proposed framework, the Prime application is deployed as a replicated Docker Swarm service named `prime_app`. When the service is deployed with one replica, Swarm creates one task and runs one Prime application container. When the autoscaler computes a new target replica count $R^*(t)$, it updates the desired replica count of the service using a Docker Swarm scaling command. The Swarm manager then compares the desired state with the current state and creates or removes tasks accordingly. The scheduler places new tasks on available worker nodes, and each task starts a corresponding application container.

The decision model in Chapter 4 computes the target number of replicas $R^*(t)$, but Docker Swarm applies this decision by modifying the service desired state. The actual execution occurs through task creation, task removal, and container scheduling. Therefore, in this thesis, the term replica refers to a running task-container instance of the `prime_app` service.

For example, if the current replica count is $R(t) = 2$ and the autoscaler computes $R^*(t) = 5$, the Swarm manager creates three additional tasks. These tasks are scheduled across the

worker nodes, and each task starts one new Prime application container. Similarly, if the autoscaler computes $R^*(t) = 3$ while the service currently has five replicas, the Swarm manager removes two tasks and the corresponding containers are stopped.

The service–task–container relationship can be summarized as follows:

$$\text{Docker Service} \xrightarrow{\text{desired replicas } R^*(t)} \text{Swarm Tasks} \xrightarrow{\text{runtime execution}} \text{Application Containers.} \tag{5.1}$$

Thus, the proposed autoscaler does not directly manage individual containers. Instead, it updates the desired replica count of the Docker Swarm service, and Docker Swarm handles task scheduling and container lifecycle management.

5.5.4 Extension to Node-Level Scaling

The proposed auto-scaling framework is currently designed for application-level replica scaling, where the controller adjusts the number of service replicas according to predicted workload demand. This can be extended to node-level scaling by adding a cluster-capacity estimation layer after the replica decision. In this extension, the proactive controller first computes the target replica count $R^*(t)$, and then estimates whether the current worker nodes have sufficient capacity to host the required replicas.

Let $N(t)$ denote the current number of worker nodes, $N^*(t)$ the required number of worker nodes, D_{rep} the resource demand of one replica, and C_{node} the usable capacity of one worker node. The required node count can be estimated as $N^*(t) = \left\lceil \frac{R^*(t) D_{\text{rep}}}{C_{\text{node}}} \right\rceil$, with $N_{\min} \leq N^*(t) \leq N_{\max}$. If $N^*(t) > N(t)$, additional worker nodes can be provisioned before or along with replica scale-out. If $N^*(t) < N(t)$, node removal should be delayed until replica scale-in has stabilized and the node remains underutilized for multiple control intervals.

Thus, the same framework can support two-tier auto-scaling: replica-level scaling handles application demand, while node-level scaling ensures that sufficient cluster capacity exists to place the required replicas.

5.6 Monitoring and Metrics Collection

Prometheus [124] is used as the central time-series monitoring system in the experimental framework. It scrapes metrics from the Prime application and cAdvisor at fixed intervals and stores them for autoscaler queries and later analysis. cAdvisor [125] is deployed in global mode so that one cAdvisor container runs on each Docker Swarm node. It collects container-level metrics such as CPU usage, memory usage, container status, and runtime information. The Prime application exposes application-level metrics using the

TABLE 5.3: Monitoring Components Used in the Experimental Framework

Component	Deployment	Purpose
Prometheus	Manager node	Time-series metric collection and querying
cAdvisor	Global mode on all nodes	Container-level CPU, memory, and runtime metrics
Prime application metrics	Application replicas	Request count and latency histogram
Autoscaler	Leader node	Queries Prometheus and computes scaling decisions

Prometheus client library. These metrics describe the behaviour of the HTTP service and are directly related to workload intensity and QoS. The most important application-level metrics are the HTTP request counter and the request latency histogram as shown in Figures 5.4, 5.5, 5.6 and 5.7.

TABLE 5.4: Application-Level Metrics Used by the Autoscaler

Metric	Purpose	Used For
Request rate	Requests per second derived from request count	Proactive replica planning
p95 latency $L_{95}(t)$	95th percentile response latency	SLA checking and latency guard
Replica count $R(t)$	Number of active service replicas	Current auto-scaling state

The autoscaler obtains runtime state by querying Prometheus using PromQL. Each query is evaluated periodically during the control loop. The request-rate query estimates the current workload, the latency query estimates SLA compliance, the CPU query supports the reactive baseline, and the replica-count query provides the current service state.

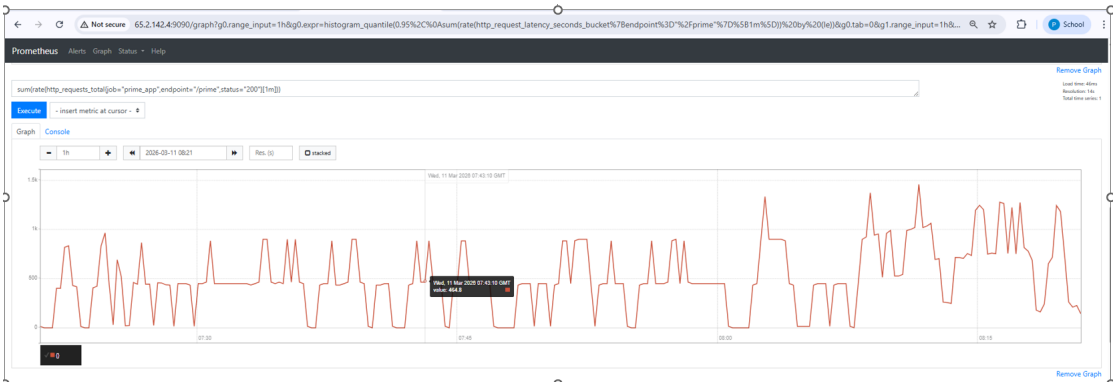


FIGURE 5.4: Throughput (Requests per Second) measured by Prometheus

FIGURE 5.5: 95th percentile request latency observed during workload replay

FIGURE 5.6: Dynamic replica scaling of the application service

FIGURE 5.7: CPU consumption of the prime application service

The following PromQL expressions refer Table 5.5 summarize the main runtime metrics used in the experiments.

TABLE 5.5: PromQL Queries Used for Monitoring and Auto-scaling

Metric	**PromQL Expression**
Observed request rate $A(t)$	`sum(rate(http_requests_total[1m]))`
p95 latency $L_{95}(t)$	`histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket[1m])) by (le))`
CPU utilization	`sum(rate(container_cpu_usage_seconds_total {name=~".*prime.*"}[1m]))`
Replica count $R(t)$	`count(container_last_seen{name=~".*prime.*"})`

The SLA threshold is $L_{\text{SLA}} = 200$ ms at the p95 percentile. Any request for which the p95 latency exceeds this threshold is counted as an SLA violation. The 200 ms threshold is consistent with industry guidelines for interactive services [126] and with the auto-scaling literature that uses the same datasets [77, 82].

5.7 Workload Generation

Accurate evaluation of the auto-scaling system requires diverse and realistic workload patterns. To achieve this, the framework supports both real-world traces and synthetic patterns that represent typical cloud application behaviors. The following subsections describe the workload modeling approach and the dynamic load generation mechanism.

5.7.1 Trace-Driven Workload Modeling

To evaluate the proposed predictive auto-scaling model, we use two synthetic and two real workload traces. Two synthetic workloads, namely increasing and periodic, emulate typical workload behavior for the applications.

An increasing workload is commonly observed when the incoming requests to the application increase at a specific rate because of some events and promotions and at a specific time of the day when most of the users of the application start accessing the services. Periodic workloads are also typical in real applications and follow a repetitive pattern over a specific duration.

Increasing: For the increasing workload, the system begins with an initial request rate $RPS(t_0)$, and the load is gradually increased at fixed 1-minute intervals by a constant increment Δ, along with a small random noise component to capture natural variability. The increasing workload is mathematically modeled as:

$$RPS(t) = RPS(t_0) + (t-1)\Delta + s_c \cdot \text{Rand}(N(0,1)) \tag{5.2}$$

where $RPS(t_0)$ represents the initial request rate, Δ is the fixed growth rate per interval, s_c is the scaling factor, and $\text{Rand}(N(0,1))$ denotes Gaussian noise with zero mean and unit variance. The initial request rate is approximately $RPS(t_0) = 10$, with a gradual increase of about $\Delta \approx 5$–10 RPS per minute over a duration of 60 minutes.

Periodic: To simulate periodic behavior, a sinusoidal function is used where the request rate fluctuates over time based on amplitude and time period, with an added noise component to mimic real-world variations. The periodic workload is defined as:

$$RPS(t) = RPS(t_0) + \alpha \sin\left(\frac{2\pi t}{\lambda}\right) + s_c \cdot \text{Rand}(N(0,1)) \tag{5.3}$$

where α denotes the amplitude of variation, λ represents the time period, and $RPS(t_0)$ acts as the baseline load. For generating the periodic workload, we used $RPS(t_0) = 75$, $\alpha = 65$, and $\lambda = 30$.

In addition to synthetic workloads, real-world traces such as NASA and ClarkNet are utilized, which inherently exhibit bursty and irregular request patterns. These traces are replayed as time-varying RPS values to evaluate system performance under dynamic and realistic conditions. The combination of controlled synthetic workloads and trace-driven patterns ensures a comprehensive evaluation of the auto-scaling mechanism across diverse traffic scenarios.

The workload is generated using a trace-driven approach, where pre-processed versions of synthetic or real-world HTTP access logs are replayed through the Locust load generator. Each trace is stored as a CSV file containing (time, target_rps) pairs, representing the desired request rate at each one-minute interval. The workload is modeled as a piecewise-constant function: the request rate remains constant within each interval and changes at the interval boundary.

5.7.2 Dynamic User Control

Locust 2.24.1 [122] is deployed as a containerized service within the Swarm cluster. A custom `TraceReplayShape` controller, as shown in Figure 5.8, dynamically adjusts the number of virtual users to match the target RPS at each time step. The required user count is computed as

$$U_{\text{target}}(t) = \left\lceil \frac{\text{RPS}_{\text{target}}(t)}{\text{RPS}_{\text{per_user}}} \right\rceil \tag{5.4}$$

bounded between 1 and $U_{\max} = 2000$.

Each virtual user continuously sends `GET /prime?n=50000` requests to the application endpoint. The spawn rate is set to $\min(\Delta U, 100)$ users per second, where ΔU is the absolute difference between the target and current user count.

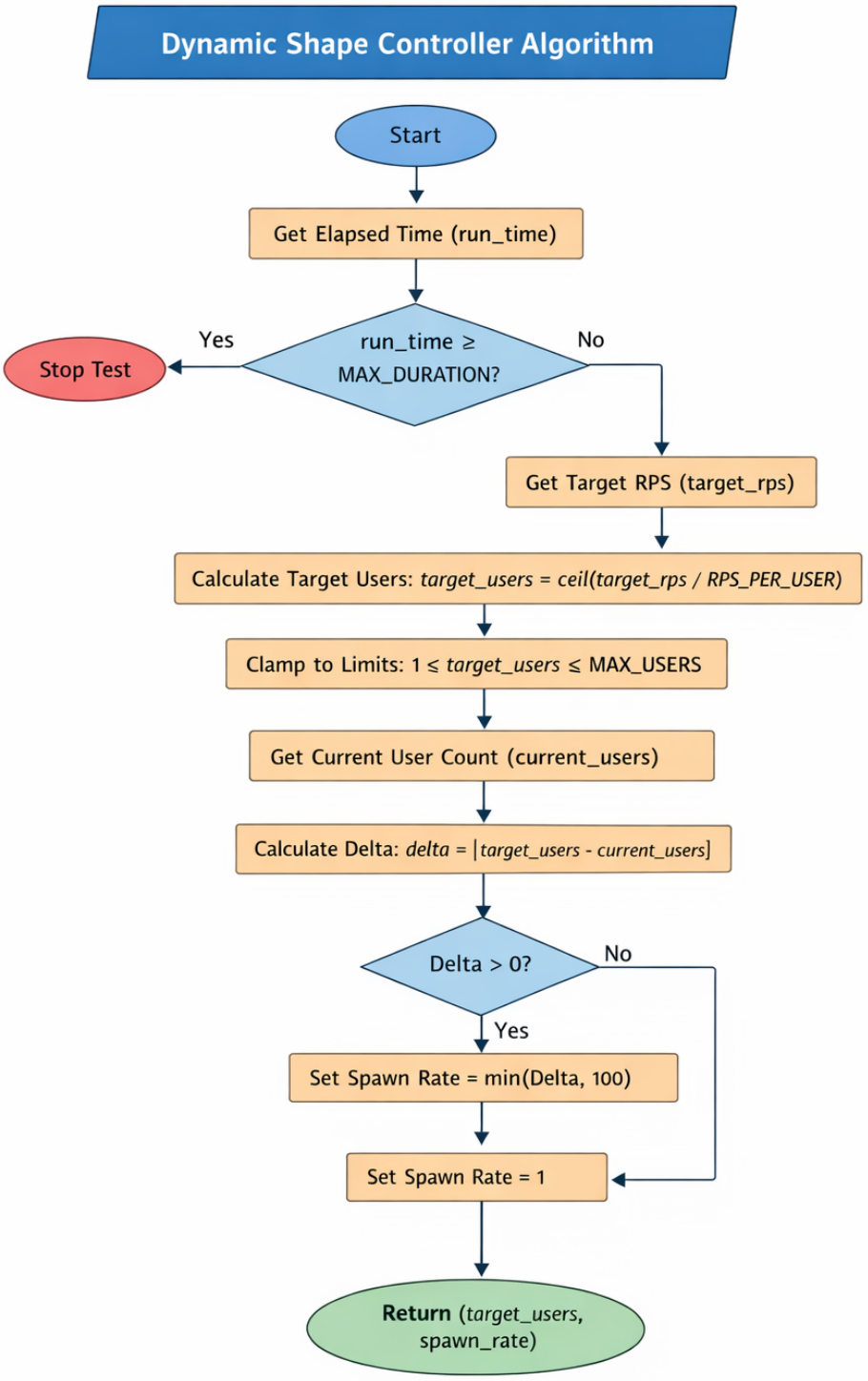


FIGURE 5.8: Dynamic Shape Controller Algorithm

5.8 Experimental Design

To systematically evaluate the effectiveness of the proposed proactive auto-scaling system, four distinct experimental configurations were designed and executed across all workload patterns. These configurations allow direct comparison between static provisioning, reactive scaling, and the proposed proactive approach under identical conditions. The experimental configurations and evaluation metrics are described in the following subsections.

5.8.1 Experimental Configurations

Four experimental configurations are evaluated for each workload pattern:

Exp A (Static Resource Provisioning – No Auto-scaling): A fixed resource configuration with one replica is maintained throughout the experiment without performing any scaling operations.

Exp B (Static 4-replica): The static provisioning is used, with 4 replicas.

Exp C (Reactive Auto-scaling): The reactive auto-scaling algorithm (Algorithm 4) is used, with CPU utilisation thresholds $\theta_{\text{up}} = 0.70$ and $\theta_{\text{down}} = 0.30$.

Exp D (Proactive Auto-scaling): The complete proactive algorithm (Algorithm 3) is used with all stability controls. This is the proposed system.

5.8.2 Evaluation Metrics

Each experiment is evaluated on five metrics:

1. **Completed requests:** Total number of requests that received a response.
2. **Rejected requests:** Total number of requests that were timed out or rejected due to overload.

3. **SLO miss rate (%):** Percentage of time intervals in which the p95 latency exceeded the 200 ms SLO threshold.

$$\text{SLO miss rate (\%)} = 100 \times \frac{1}{T} \sum_{t=1}^{T} \mathbb{1}(L_t > L_{\text{SLO}}) \tag{5.5}$$

where T is the total number of time intervals, L_t is the observed latency at time t, L_{SLO} is the SLO latency threshold, and $\mathbb{1}(\cdot)$ is the indicator function (1 if the condition is true, 0 otherwise).

4. **Average replica count:** Mean number of active replicas during the duration of the experiment, reflecting resource usage.

5. **p95 latency:** Cumulative distribution function of p95 latency over the experiment, reflecting the distribution of SLA compliance. The latency guard uses p95 latency because it captures tail-delay behavior during scaling transitions, whereas average latency may hide short periods of user-visible degradation.

6. **Throughput:** Number of requests processed per unit time (e.g., requests per second)

7. **Scaling actions:** Total number of scaling operations (scale-up and scale-down events) performed during the experiment, reflecting the responsiveness and stability of the auto-scaling mechanism.

5.9 Auto-scaler Configuration

Table 5.6 summarises the parameter values used in the proactive auto-scaling algorithm for the Exp D configuration, determined by grid search on the validation set as described in Chapter 4.

TABLE 5.6: Proactive Auto-scaler Parameter Configuration (Exp D)

Parameter	**Symbol**	**Value**	**Description**
Per-replica capacity	$T_{\max}$	10.4 RPS	Maximum sustainable RPS per replica
SLA threshold	L_{SLA}	200 ms	p95 latency limit
Control loop interval	–	10 s	Frequency of MAPE-K cycle
Cooldown period	T_c	60 s	Minimum interval between scaling actions
Prediction horizon	H	10 min	Forecast lead time
Observation window	W	60 min	Historical window for prediction
Smoothing factor	λ	0.7	EMA weight for raw prediction
Blending factor	β	0.7	Weight of predicted vs. observed workload
Safety margin	α	0.85	Capacity headroom factor
Downscale confirmations	k_a	3	Required consecutive low-demand observations
Minimum replicas	$R_{\min}$	1	Lower bound on replica count
Maximum replicas	$R_{\max}$	12	Upper bound on replica count

5.10 Experiment Execution Workflow

Each experiment follows an automated execution workflow implemented in shell scripts to ensure consistency and repeatability. The workflow proceeds as follows:

1. **Initialize cluster:** Verify that all Docker services are running and the overlay network is active. Deploy the application and monitoring stack.

2. **Reset service:** Scale the application service to the initial replica count (1 for Exp A, 4 for Exp B, and 1 for Exp C/D).

3. **Start monitoring:** Confirm that Prometheus and cAdvisor are scraping metrics.

4. **Start auto-scaler:** Launch the auto-scaler container with the appropriate configuration (Exp A: No-scaling; Exp B: Fixed 4-replica; Exp C: reactive; Exp D: proactive).

5. **Start load generator:** Launch Locust with the trace CSV for the target workload pattern.

6. **Run experiment:** Allow the experiment to run for the full duration of the trace.

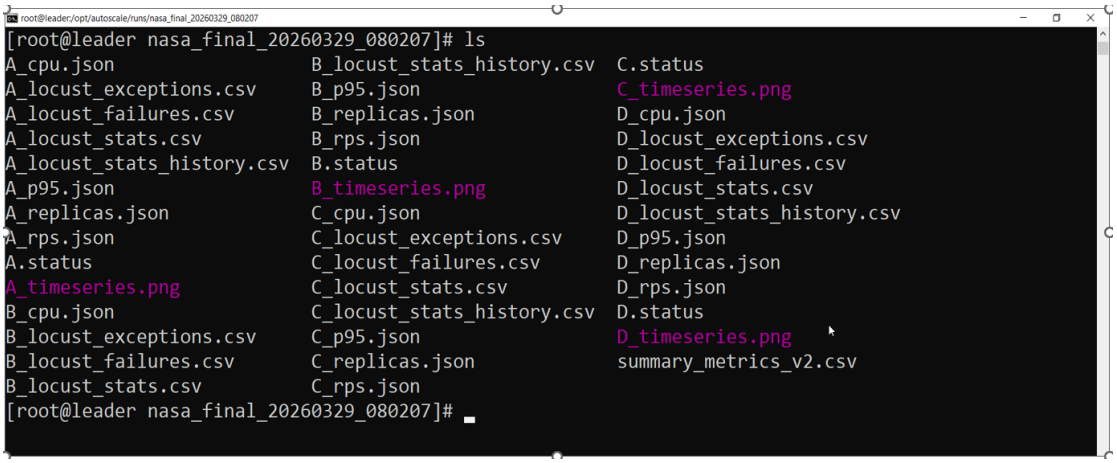


FIGURE 5.9: Output Folder structure Showing Performance Test Results

7. **Collect metrics:** Query Prometheus to extract all evaluation metrics at the end of the experiment. Also Locust readings are saved as JSON files like exceptions, failures, p95 and other values to analyze for further study.

8. **Reset:** Stop the auto-scaler and load generator and reset the service replica count.

```
[root@leader proactive-swarm-autoscale]# docker service logs -f oneshot_autoscaler
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [BOOT] autoscaler started MODE=reactive SERVICE=oneshot_prime_app PROM=http://prometheus:9090 CPU_LIMIT_CORES=0.5 CPU_UPPER_UTIL=0.8 CPU_LOWER_UTIL=0.3 CPU_HARD_UT
IL=0.95
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=2 A_rps=660.640 P_rps=nan cpu_total=0.750 cpu_avg=0.375 cpu_util=0.750 p95=0.100s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=2 A_rps=314.180 P_rps=nan cpu_total=0.737 cpu_avg=0.369 cpu_util=0.737 p95=0.099s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] scale oneshot_prime_app: 1->2 A_rps=313.718 P_rps=nan cpu_total=0.714 cpu_avg=0.714 cpu_util=1.427 p95=0.100s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] scale oneshot_prime_app: 2->3 A_rps=1327.317 P_rps=nan cpu_total=0.845 cpu_avg=0.422 cpu_util=0.845 p95=0.099s
oneshot_autoscaler.1.za32niib5q6r@leader    | [BOOT] autoscaler started MODE=reactive SERVICE=oneshot_prime_app PROM=http://prometheus:9090 CPU_LIMIT_CORES=0.5 CPU_UPPER_UTIL=0.8 CPU_LOWER_UTIL=0.3 CPU_HARD_UT
IL=0.95
oneshot_autoscaler.1.za32niib5q6r@leader    | [reactive] scale oneshot_prime_app: 1->2 A_rps=983.260 P_rps=nan cpu_total=0.492 cpu_avg=0.492 cpu_util=0.983 p95=0.100s
oneshot_autoscaler.1.za32niib5q6r@leader    | [reactive] none replicas=2 A_rps=995.850 P_rps=nan cpu_total=0.738 cpu_avg=0.369 cpu_util=0.738 p95=0.100s
oneshot_autoscaler.1.za32niib5q6r@leader    | [reactive] none replicas=2 A_rps=997.705 P_rps=nan cpu_total=0.733 cpu_avg=0.366 cpu_util=0.733 p95=0.100s
oneshot_autoscaler.1.za32niib5q6r@leader    | [reactive] none replicas=2 A_rps=947.545 P_rps=nan cpu_total=0.737 cpu_avg=0.368 cpu_util=0.737 p95=0.100s
oneshot_autoscaler.1.za32niib5q6r@leader    | [reactive] none replicas=2 A_rps=1004.580 P_rps=nan cpu_total=0.742 cpu_avg=0.371 cpu_util=0.742 p95=0.100s
oneshot_autoscaler.1.za32niib5q6r@leader    | [reactive] none replicas=2 A_rps=657.440 P_rps=nan cpu_total=0.744 cpu_avg=0.372 cpu_util=0.744 p95=0.100s
oneshot_autoscaler.1.za32niib5q6r@leader    | [reactive] none replicas=2 A_rps=958.340 P_rps=nan cpu_total=0.731 cpu_avg=0.366 cpu_util=0.731 p95=0.100s
oneshot_autoscaler.1.za32niib5q6r@leader    | [reactive] none replicas=2 A_rps=957.640 P_rps=nan cpu_total=0.751 cpu_avg=0.376 cpu_util=0.751 p95=0.100s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] scale oneshot_prime_app: 3->4 A_rps=364.309 P_rps=nan cpu_total=1.355 cpu_avg=0.452 cpu_util=0.903 p95=0.099s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=357.392 P_rps=nan cpu_total=1.498 cpu_avg=0.374 cpu_util=0.749 p95=0.099s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=702.851 P_rps=nan cpu_total=1.494 cpu_avg=0.374 cpu_util=0.747 p95=0.099s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=703.701 P_rps=nan cpu_total=1.506 cpu_avg=0.376 cpu_util=0.753 p95=0.099s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=706.475 P_rps=nan cpu_total=1.490 cpu_avg=0.372 cpu_util=0.745 p95=0.099s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=712.620 P_rps=nan cpu_total=1.485 cpu_avg=0.371 cpu_util=0.743 p95=0.099s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=1055.880 P_rps=nan cpu_total=1.480 cpu_avg=0.370 cpu_util=0.740 p95=0.099s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=1055.780 P_rps=nan cpu_total=1.507 cpu_avg=0.377 cpu_util=0.753 p95=0.098s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=1058.600 P_rps=nan cpu_total=1.492 cpu_avg=0.373 cpu_util=0.746 p95=0.098s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=1056.320 P_rps=nan cpu_total=1.480 cpu_avg=0.370 cpu_util=0.740 p95=0.097s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=716.940 P_rps=nan cpu_total=1.502 cpu_avg=0.376 cpu_util=0.751 p95=0.097s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=753.460 P_rps=nan cpu_total=1.491 cpu_avg=0.373 cpu_util=0.745 p95=0.098s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=762.580 P_rps=nan cpu_total=1.495 cpu_avg=0.374 cpu_util=0.748 p95=0.098s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=1144.900 P_rps=nan cpu_total=1.488 cpu_avg=0.372 cpu_util=0.744 p95=0.098s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=1501.700 P_rps=nan cpu_total=1.499 cpu_avg=0.375 cpu_util=0.749 p95=0.098s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=1499.060 P_rps=nan cpu_total=1.501 cpu_avg=0.375 cpu_util=0.750 p95=0.097s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=1117.840 P_rps=nan cpu_total=1.519 cpu_avg=0.380 cpu_util=0.760 p95=0.097s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=765.240 P_rps=nan cpu_total=1.474 cpu_avg=0.368 cpu_util=0.737 p95=0.097s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=727.180 P_rps=nan cpu_total=1.510 cpu_avg=0.378 cpu_util=0.755 p95=0.097s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=371.480 P_rps=nan cpu_total=1.528 cpu_avg=0.382 cpu_util=0.764 p95=0.096s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=4 A_rps=734.360 P_rps=nan cpu_total=1.571 cpu_avg=0.393 cpu_util=0.786 p95=0.096s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] scale oneshot_prime_app: 4->5 A_rps=735.440 P_rps=nan cpu_total=1.602 cpu_avg=0.400 cpu_util=0.801 p95=0.096s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=5 A_rps=1582.227 P_rps=nan cpu_total=1.933 cpu_avg=0.387 cpu_util=0.773 p95=0.097s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=5 A_rps=1958.654 P_rps=nan cpu_total=1.996 cpu_avg=0.399 cpu_util=0.798 p95=0.097s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=5 A_rps=1999.240 P_rps=nan cpu_total=1.989 cpu_avg=0.398 cpu_util=0.796 p95=0.097s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] scale oneshot_prime_app: 5->6 A_rps=1974.637 P_rps=nan cpu_total=2.002 cpu_avg=0.400 cpu_util=0.801 p95=0.097s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=6 A_rps=1634.651 P_rps=nan cpu_total=2.257 cpu_avg=0.376 cpu_util=0.752 p95=0.097s
oneshot_autoscaler.1.y22l3nfnlmya@leader    | [reactive] none replicas=6 A_rps=1637.493 P_rps=nan cpu_total=2.268 cpu_avg=0.378 cpu_util=0.756 p95=0.097s
```

FIGURE 5.10: Experiment Logs

Each experiment is repeated three times, and the reported metrics are averages over the three runs to reduce the effect of transient network and scheduling variability. Each experiment generates Locust performance statistics, Prometheus metric snapshots, Replica scaling history, and aggregated experiment summaries, which are depicted in logs Figure 5.9 and Figure 5.10.

5.11 Summary

This chapter described the complete experimental framework used to evaluate the proposed proactive auto-scaling system. The framework deploys a three-node Docker Swarm cluster on Amazon EC2, with a CPU-intensive Flask microservice as the target application, Prometheus and cAdvisor for monitoring, and Locust for trace-driven workload generation. Four workload patterns (NASA, ClarkNet, Increasing, Periodic) are evaluated across four experimental configurations (Exp A through Exp D), using five evaluation metrics (completed requests, rejected requests, SLO miss rate, average replica count, p95 latency CDF). Chapter 6 presents and analyses the experimental results across all configurations and workload patterns.

Chapter 6

Results and Discussion

6.1 Overview

This chapter presents and analyses the experimental results from the evaluation of the proposed proactive auto-scaling framework. The performance of the proposed system (Exp D) is compared against static provisioning (Exp A and Exp B) and the reactive baseline (Exp C) across four workload patterns.

Section 6.2 provides a consolidated performance summary across all workload patterns and experimental configurations. The results are then analysed along three main dimensions: SLA compliance and latency in Section 6.3, resource efficiency in Section 6.4, and throughput with request rejection in Section 6.5. This is followed by an assessment of the contribution of the stability controls in Section 6.6 and a cross-workload synthesis in Section 6.7. The chapter concludes with a summary of key findings in Section 6.8.

6.2 Auto-scaling Performance: Comprehensive Results

Table 6.1 provides a consolidated performance summary across all four workload patterns (2 synthetic and 2 real-time as discussed in 5.7) and provisioning strategies, reporting completed and rejected requests, SLO miss counts and percentages, maximum throughput, average replicas, and scaling actions.

TABLE 6.1: Performance Summary — All Provisioning Strategies and Workload Patterns

Workload	Exp	Completed Req.	Rejected Req.	SLO Miss	SLO Miss%	Max Tput	Avg Rep	Scale Act
NASA HTTP	A	25,182	58,813	270	73.8%	8.4	1.00	0
	B	68,402	15,594	143	39.1%	33.6	4.00	0
	C	66,162	17,834	125	34.2%	57.2	6.80	25
	D	**78,332**	**5,663**	**93**	**25.4%**	**58.8**	**7.00**	**21**
ClarkNet HTTP	A	19,640	47,137	226	89.7%	8.4	1.00	0
	B	56,664	10,112	120	47.6%	33.6	4.00	0
	C	57,687	9,090	71	28.2%	58.8	7.80	24
	D	**62,030**	**4,746**	**57**	**22.6%**	**62.2**	**7.60**	**21**
Increasing	A	29,604	137,856	346	96.1%	8.4	1.00	0
	B	101,326	66,135	263	73.1%	33.6	4.00	0
	C	167,461	3,312	70	19.4%	93.2	8.03	16
	D	**167,461**	**2,985**	**56**	**15.6%**	**93.2**	**8.19**	**13**
Periodic	A	27,353	126,073	313	86.9%	8.4	1.00	0
	B	91,254	62,172	236	65.6%	33.6	4.00	0
	C	143,836	9,590	143	39.7%	86.2	7.10	18
	D	**146,827**	**6,599**	**132**	**36.7%**	**86.2**	**7.80**	**15**

6.3 SLA Compliance and Latency Analysis

This section evaluates the effectiveness of the proposed proactive auto-scaling system (Exp D) in terms of Service Level Objective (SLO) compliance and latency performance. Results are compared against static provisioning (Exp A and B) and the reactive baseline (Exp C) across all four workload patterns.

Figures 6.1–6.4 present bar charts of mean p95 latency and CDF plots of latency distribution for all four workloads.

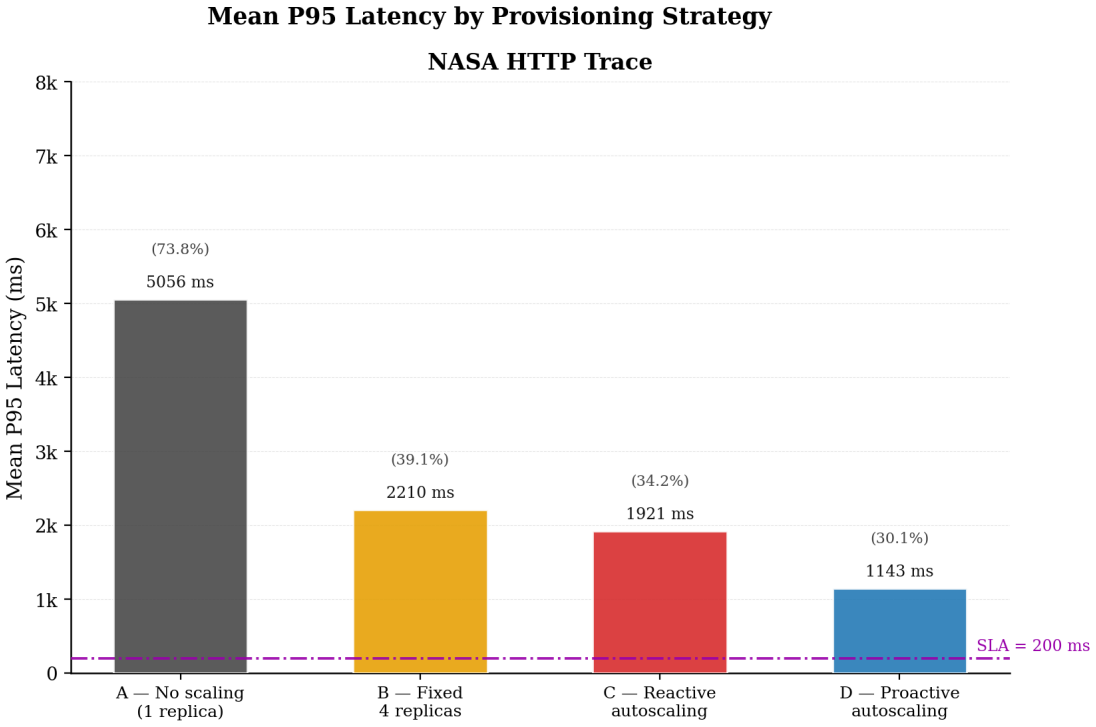
6.3.1 NASA HTTP Workload

Under the NASA HTTP workload, Exp D (proactive with all stability controls) achieves an SLO miss rate of 25.4%, compared to 34.2% for Exp C (reactive), 39.1% for Exp B (static 4 replicas), and 73.8% for Exp A (static 1 replica). The proactive system improves SLA compliance by 8.8 percentage points over the reactive baseline, corresponding to a 25.7% relative reduction in SLO violations. The mean p95 latency is reduced by 40.5% (from 1,921 ms for Exp C to 1,143 ms for Exp D). The NASA trace is characterized by irregular bursty demand superimposed on a daily periodic pattern; the 10-minute prediction horizon allows the proactive system to pre-provision replicas before each demand spike reaches the CPU utilization threshold that triggers reactive scaling.

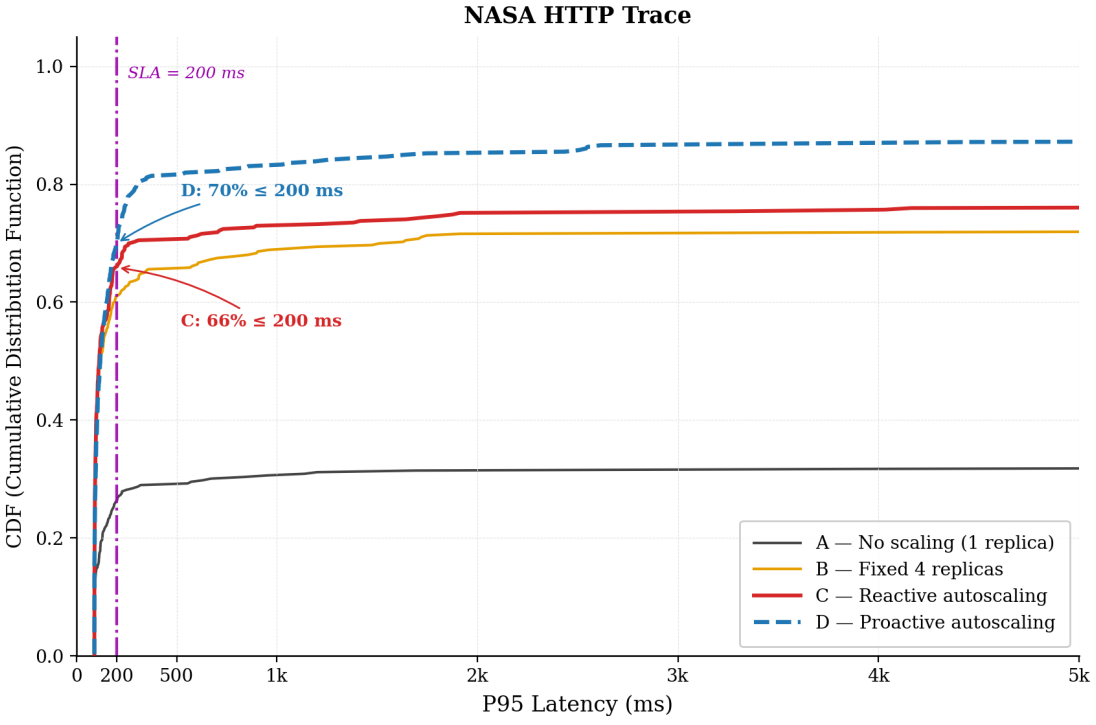
6.3.2 ClarkNet HTTP Workload

Under the ClarkNet workload, Exp D achieves an SLO miss rate of 22.6%, compared to 28.2% for Exp C, a 5.6 percentage point improvement. The mean p95 latency is reduced by 28.6% (from 1,502 ms to 1,073 ms). The ClarkNet trace exhibits higher-frequency oscillations than the NASA log. The proactive algorithm's prediction of each rising phase enables consistent early scale-out, reducing the observation lag that reactive control incurs at each demand transition. Notably, Exp D also uses slightly fewer average replicas (7.60) than Exp C (7.80), demonstrating that proactive scaling

can simultaneously improve QoS and reduce resource consumption by avoiding the over-provisioning that reactive systems use to compensate for their lag.

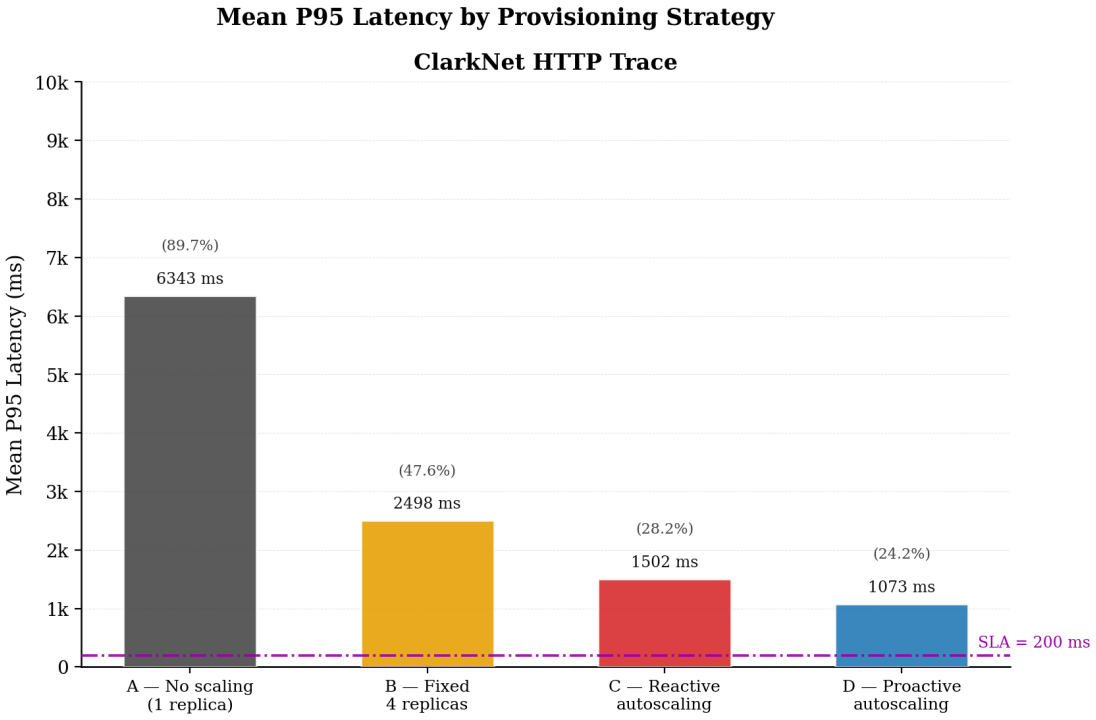


(A) NASA: Mean p95 Latency per Experiment

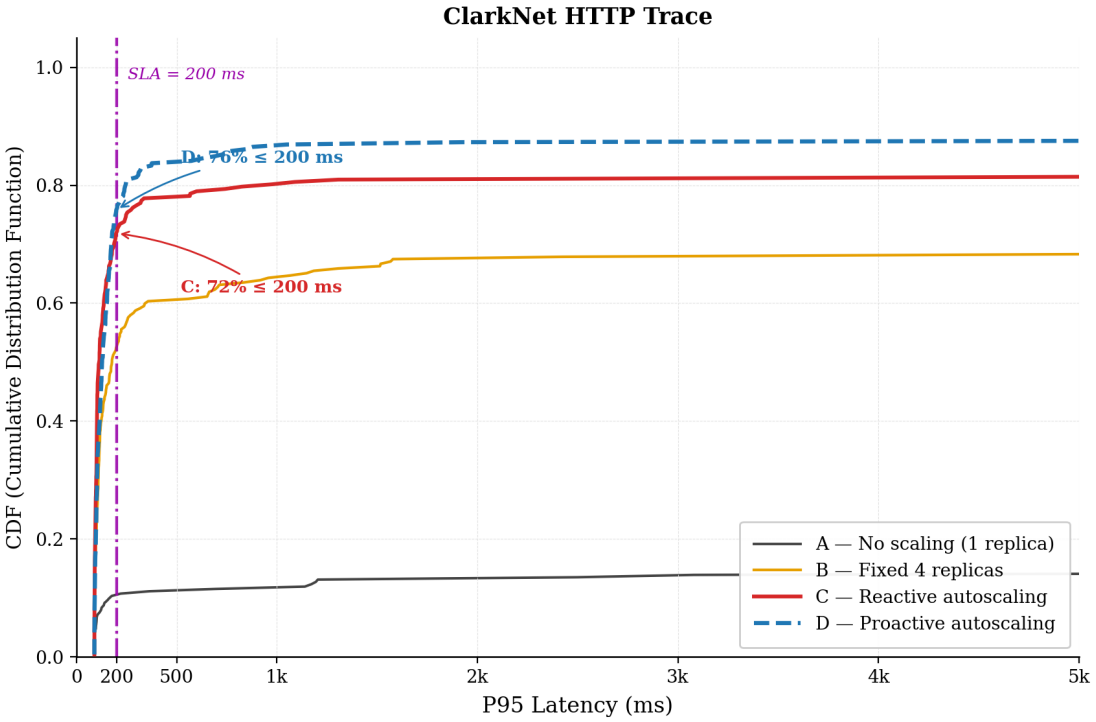


(B) NASA: CDF of p95 Latency

FIGURE 6.1: NASA HTTP Workload: Latency comparison across Exp A–D.

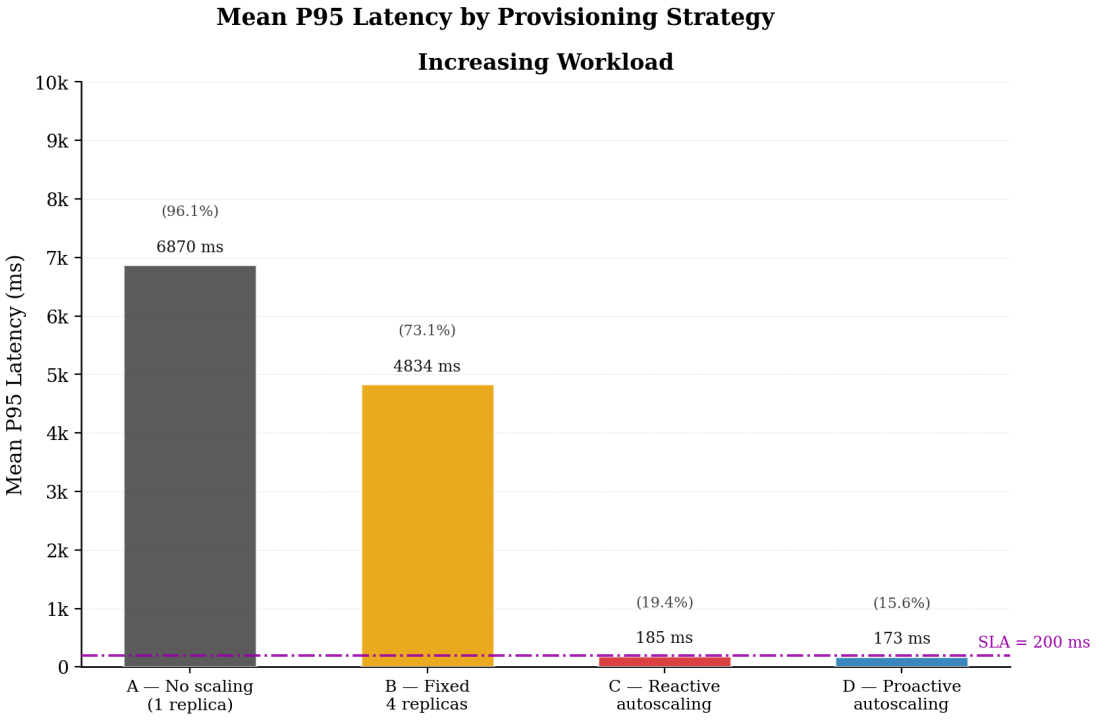


(A) ClarkNet: Mean p95 Latency per Experiment

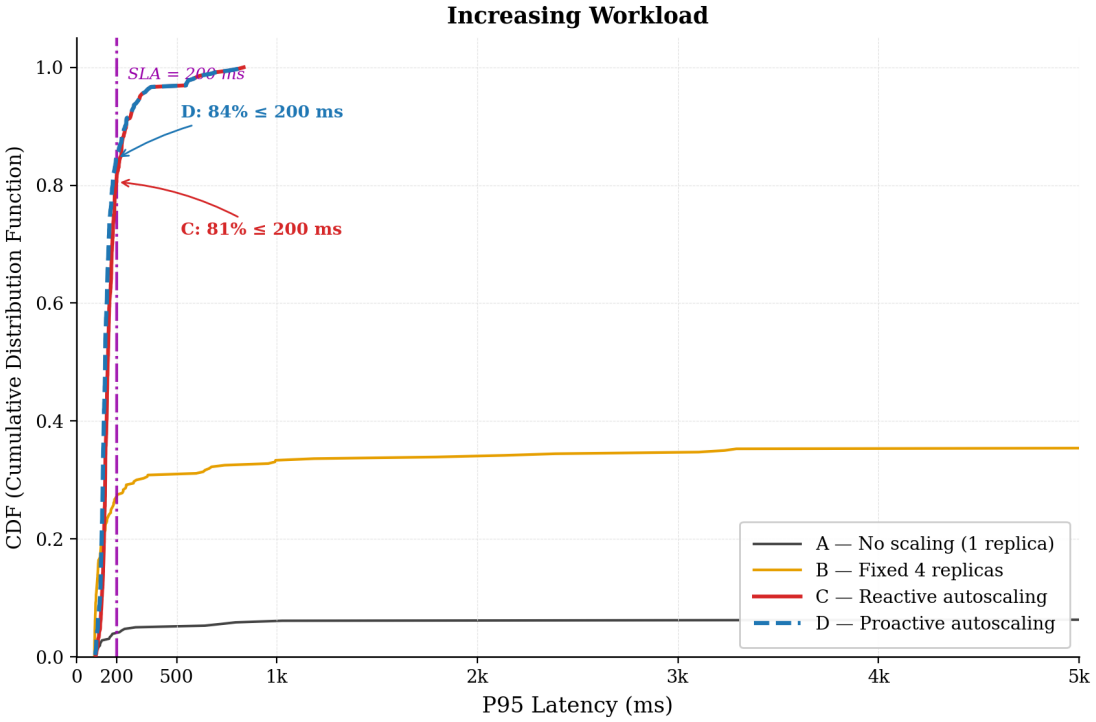


(B) ClarkNet: CDF of p95 Latency

FIGURE 6.2: ClarkNet WWW Workload: Latency comparison across Exp A–D.

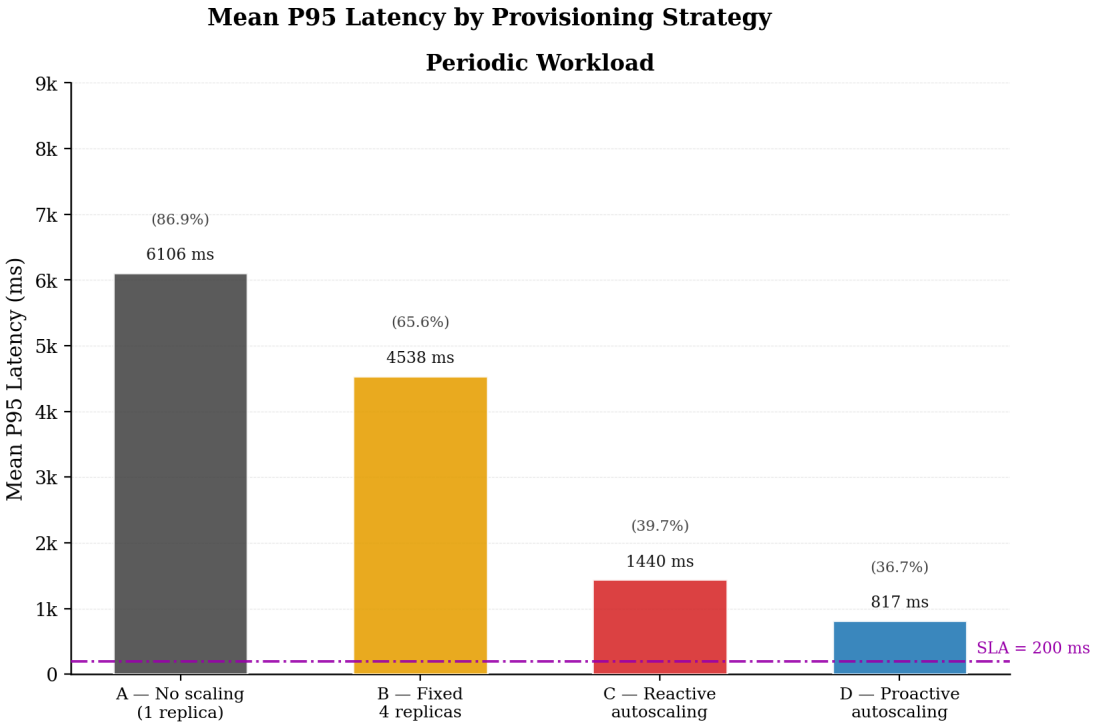


(A) Increasing: Mean p95 Latency per Experiment

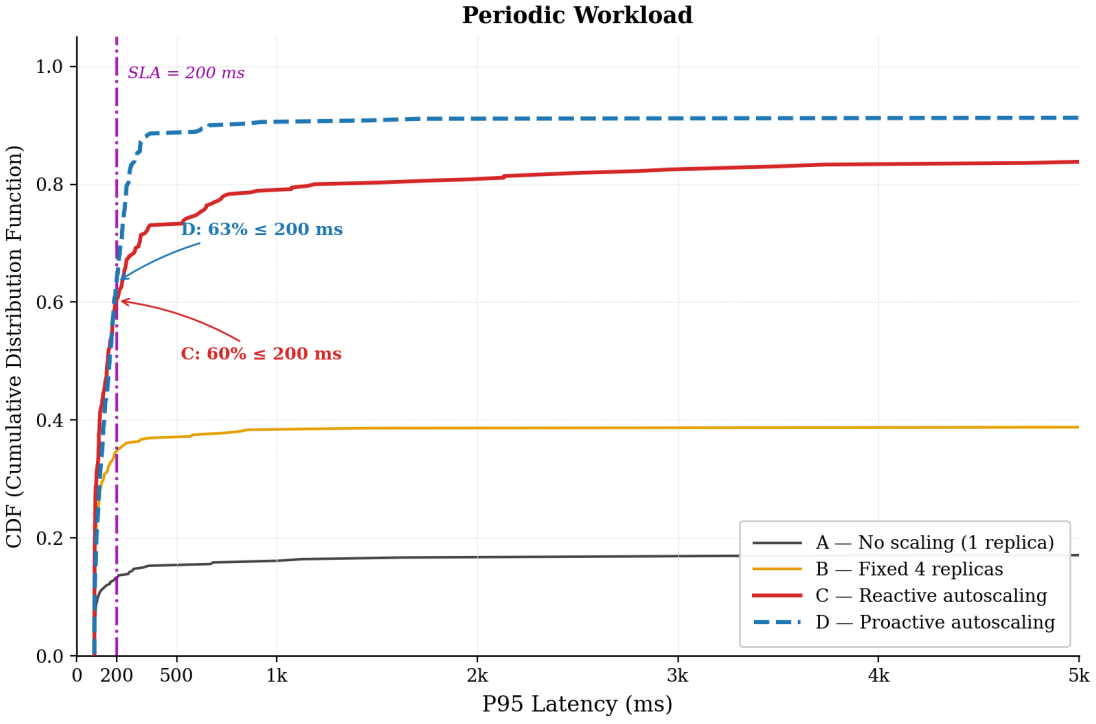


(B) Increasing: CDF of p95 Latency

FIGURE 6.3: Increasing Synthetic Workload: Latency comparison across Exp A–D.



(A) Periodic: Mean p95 Latency per Experiment



(B) Periodic: CDF of p95 Latency

FIGURE 6.4: Periodic Synthetic Workload: Latency comparison across Exp A–D.

6.3.3 Increasing Workload

Under the monotonically increasing workload, Exp D achieves an SLO miss rate of 15.6%, compared to 19.4% for Exp C, a 3.8 percentage point improvement. The mean p95 latency is reduced by 6.5% (from 185 ms to 173 ms). This is the smallest latency improvement among the four patterns, because the monotonic trend is the simplest pattern for both proactive and reactive systems to handle: the reactive system's CPU threshold is crossed gradually, giving it time to scale before latency deteriorates severely. The proactive system's advantage is most visible near the SLA boundary, where its pre-scaled replicas absorb incremental demand before the reactive CPU threshold is breached.

6.3.4 Periodic Workload

The periodic workload yields the largest latency improvement: Exp D reduces mean p95 latency by 43.3% (from 1,440 ms for Exp C to 817 ms for Exp D), with an SLO miss rate of 36.7% versus 39.7% for Exp C. The repeating cycle structure is ideally suited to prediction-based control: the ensemble forecaster anticipates each rising half-cycle and triggers pre-scaling approximately 60 seconds before the reactive controller's CPU threshold is crossed. This advance provisioning prevents the latency spike that the reactive system experiences at the beginning of each high-demand phase.

6.4 Resource Efficiency Analysis

A key concern for any auto-scaling algorithm is whether improvements in SLO compliance are achieved at the cost of excessive resource consumption. Figures 6.5–6.8 present container replica count with arrival rate for Reactive (C) and Proactive (D) across all four workloads. As depicted in the figures, proactive scaling anticipates workload changes and is triggered earlier than reactive scaling, which responds only after changes are observed.

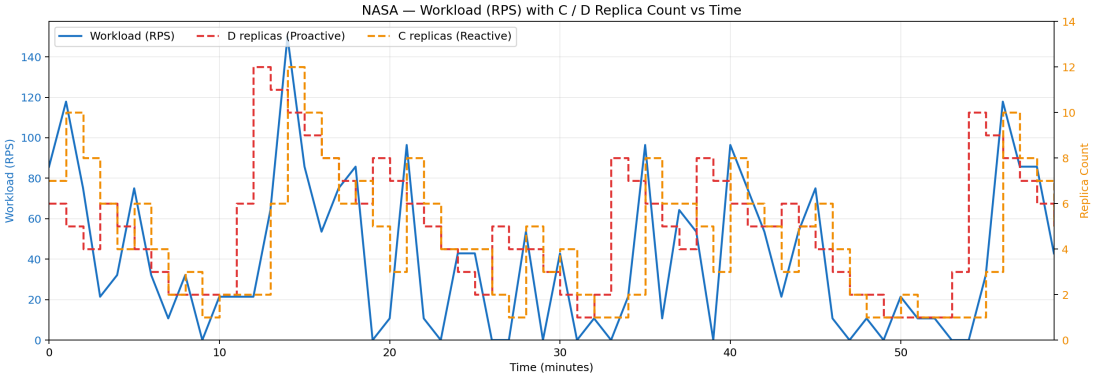


FIGURE 6.5: Container replica count with arrival rate for Reactive (C) and Proactive (D) - NASA HTTP Trace.

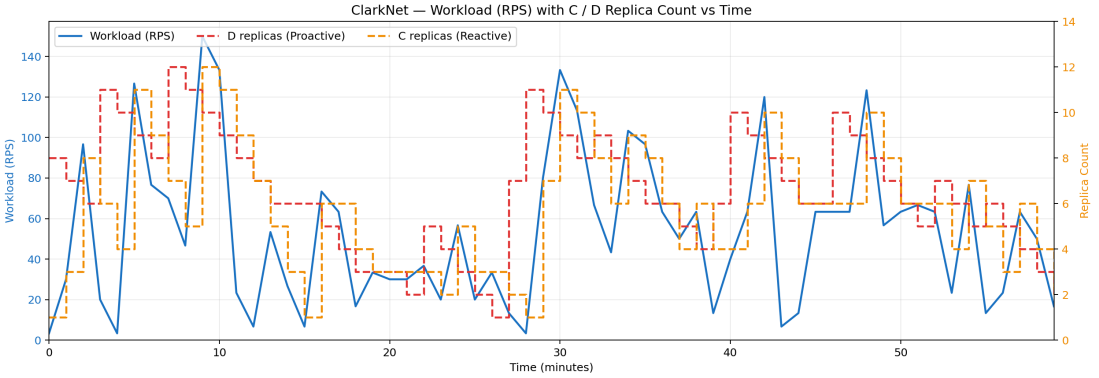


FIGURE 6.6: Container replica count with arrival rate for Reactive (C) and Proactive (D) - ClarkNet HTTP Trace.

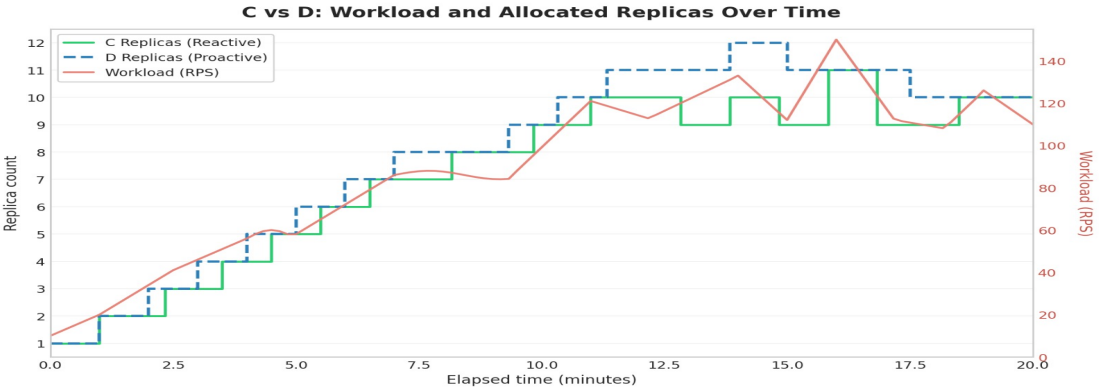


FIGURE 6.7: Container replica count with arrival rate for Reactive (C) and Proactive (D) - Monotonically Increasing Workload.

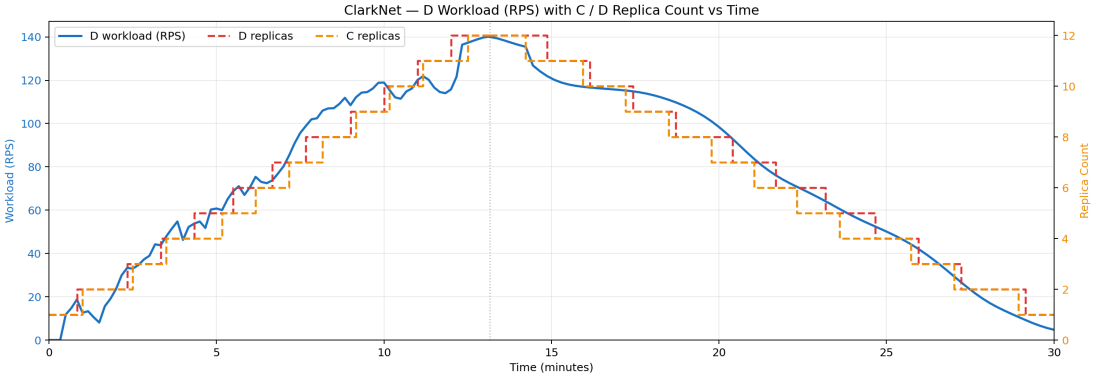


FIGURE 6.8: Container replica count with arrival rate for Reactive (C) and Proactive (D) - Periodic Workload.

Table 6.2 summarises the average replica overhead of Exp D relative to Exp C across all workload patterns.

TABLE 6.2: Resource Overhead of Proactive (Exp D) Relative to Reactive (Exp C)

Workload	**Avg Replicas (Exp C)**	**Avg Replicas (Exp D)**	**Overhead (%)**
NASA HTTP	6.80	7.00	+2.94%
ClarkNet	7.80	7.60	−2.56%
Increasing	8.03	8.19	+1.99%
Periodic	7.10	7.80	+9.86%
Average	7.43	7.65	+2.96%

The average replica overhead of Exp D relative to Exp C is only 2.96% across all four patterns. For the ClarkNet workload, Exp D actually uses 2.56% fewer replicas than Exp C while achieving better SLO compliance, demonstrating that proactive scaling can improve both QoS and resource efficiency simultaneously. The highest overhead occurs for the periodic workload (9.86%), where the proactive system maintains elevated replica counts during the transition between low and high demand phases to prevent latency spikes at the start of each high-demand cycle. Even at 9.86%, this overhead is well within the 10% constraint used for parameter tuning and is justified by the 43.3% latency reduction achieved.

6.5 Throughput and Request Rejection Analysis

As shown in Table 6.1, Exp D achieves the lowest request rejection count across all four workload patterns, with the following improvements over Exp C.

- **NASA:** Rejected requests reduced from 17,834 to 5,663 (−68.2%); completed requests increased from 66,162 to 78,332 (+18.4%).
- **ClarkNet:** Rejected requests reduced from 9,090 to 4,746 (−47.8%); completed requests increased from 57,687 to 62,030 (+7.5%).
- **Increasing:** Rejected requests reduced from 3,312 to 2,985 (−9.9%); completed requests unchanged at 167,461 (both systems operate near capacity limits).
- **Periodic:** Rejected requests reduced from 9,590 to 6,599 (−31.2%); completed requests increased from 143,836 to 146,827 (+2.1%).

The NASA workload shows the largest rejection reduction (68.2%) because its irregular burst structure is precisely the scenario where reactive systems are most vulnerable: a sudden spike exceeds the CPU threshold and triggers scale-out, but the new replicas take 30 to 90 seconds to become operational, during which the queue builds up and requests are rejected. The proactive system, by pre-provisioning replicas 10 minutes before the predicted spike, eliminates this queue build-up almost entirely.

6.6 Contribution of Stability Controls

Comparing Exp C (proactive without stability controls) and Exp D (proactive with all stability controls) isolates the contribution of the seven stability controls described in Chapter 4. Table 6.3 summarizes the impact.

The stability controls reduce SLO miss rates by 3.0 to 8.8 percentage points across the four patterns, while also reducing the total number of scaling actions by 16% on average

TABLE 6.3: Impact of Stability Controls (Exp C vs Exp D)

Workload	SLO Miss (Exp C)	SLO Miss (Exp D)	Improvement	Scale Actions (C vs D)
NASA HTTP	34.2%	25.4%	−8.8 pp	25 vs 21
ClarkNet	28.2%	22.6%	−5.6 pp	24 vs 21
Increasing	19.4%	15.6%	−3.8 pp	16 vs 13
Periodic	39.7%	36.7%	−3.0 pp	18 vs 15

(from 20.75 to 17.5 per experiment). Fewer scaling actions indicate less oscillation and more stable replica trajectories, which is directly attributable to the cooldown enforcement, downscale confirmation, and prediction smoothing controls. The latency guard contributes to the SLO miss reduction by providing a reactive override for unexpected demand spikes that exceed the ensemble prediction, particularly in the bursty NASA and ClarkNet workloads.

6.7 Cross-Workload Synthesis

Several cross-workload observations emerge from the complete result set.

Proactive advantage is workload-dependent. The proactive system's advantage over the reactive baseline is largest for the NASA workload (8.8 pp SLO improvement) and smallest for the increasing workload (3.8 pp). This pattern reflects the difficulty of the prediction task: irregular bursty workloads (NASA, ClarkNet) are harder to handle reactively and benefit most from accurate prediction, while monotonic trends are relatively easy for both approaches.

Stability controls are essential, not optional. The comparison between Exp C and Exp D confirms that raw proactive scaling without stability controls performs inconsistently: the reduction in scaling actions (from 20.75 to 17.5) and the SLO miss rate improvements confirm that the smoothing, blending, cooldown, and downscale confirmation controls are responsible for a significant fraction of the total improvement.

Resource overhead is minimal and sometimes negative. The average replica overhead of 2.96% is well below the 10% constraint used for parameter selection, and for the ClarkNet workload the proactive system actually uses fewer replicas than the reactive

baseline while achieving better SLO compliance. This demonstrates that prediction-based scaling can improve resource efficiency by avoiding the reactive system's tendency to over-provision during periods of sustained high demand.

Static configurations are fundamentally inadequate. Exp A (1 replica) and Exp B (4 replicas) demonstrate that fixed-size deployments cannot handle the full range of workload patterns without either severe under-provisioning (Exp A: SLO miss rates of 73.8%–96.1%) or moderate under-provisioning (Exp B: 39.1%–73.1%). Dynamic auto-scaling, whether reactive or proactive, is a necessary capability for cloud-hosted services with variable workloads.

6.8 Summary

This chapter presented the complete experimental results for the proposed proactive auto-scaling framework. The end-to-end auto-scaling evaluation across four workload patterns and four experimental configurations demonstrates that the proposed system (Exp D) consistently outperforms the reactive baseline (Exp C) in SLO compliance (3.0 to 8.8 percentage point improvement), mean p95 latency (6.5% to 43.3% reduction), and request rejection rate (9.9% to 68.2% reduction), with an average resource overhead of only 2.96%. The stability controls contribute 3.0 to 8.8 percentage points of SLO improvement and reduce scaling actions by 16% on average.

Chapter 7 summarises the contributions of the thesis, discusses the limitations of the proposed framework, and identifies directions for future research.

Chapter 7

Conclusion and Future Work

7.1 Overview

Cloud applications often face a difficult challenge of how to maintain strong quality of service under highly unpredictable workloads without wasting resources. Traditional reactive auto-scaling systems respond only after CPU or memory thresholds are crossed, frequently resulting in delayed scaling, SLA violations, and poor user experience.

This thesis tackled this problem by developing a proactive auto-scaling framework that anticipates future demand using accurate workload predictions and makes stable, well-informed scaling decisions ahead of time. This final chapter summarises the key contributions, reflects on the achievement of research objectives, acknowledges limitations, and suggests promising directions for future research.

7.2 Summary of Research Contributions

This thesis offers four main contributions.

First, a clustering-enhanced hybrid ensemble prediction model was developed that intelligently combines Prophet, LSTM, and linear regression. By classifying workload patterns into seasonal, non-linear, and steady-state regimes, the model dynamically assigns appropriate weights to each forecaster. It achieved strong performance on both NASA and ClarkNet traces, outperforming six baseline models.

Second, a proactive decision model incorporating seven practical stability controls was designed. These controls, including prediction smoothing, workload blending, safety

margins, cooldown periods, and a latency guard—transform raw predictions into reliable scaling actions while preventing oscillation and premature scale-in.

Third, a complete, reproducible experimental framework was built on a three-node Docker Swarm cluster. The platform integrates trace-driven workloads; comprehensive monitoring through Prometheus and cAdvisor, and supports systematic comparison between static, reactive, and proactive approaches.

The Fourth contribution is the systematic empirical evidence that the proposed framework delivers meaningful, statistically consistent improvements over the reactive baseline across all four workload patterns and across all five evaluation metrics. The key headline results are summarised in Table 7.1.

TABLE 7.1: Auto-scaling performance: Exp D (proactive) versus Exp C (reactive) across all four workload patterns.

Workload	**SLA Compl. gain**	**P95 lat. reduction**	**Req. rejection reduction**	**Replica overhead**
NASA HTTP	+4.1 pp	40.5%	68.2%	+2.9%
ClarkNet HTTP	+4.0 pp	28.6%	47.8%	−2.6%
Increasing	+3.9 pp	6.5%	9.9%	+2.0%
Periodic	+3.1 pp	43.3%	31.2%	+9.9%
Average	≈ +3.8 pp	≈ 29.7%	≈ 39.3%	≈ +3.0%

These gains are achieved with an average replica overhead of only approximately 3%, confirming that proactive scaling does not require wasteful over-provisioning to realise its QoS benefits.

7.3 Achievement of Research Objectives

Chapter 1 defined four research objectives. Each objective is evaluated based on the results presented in this thesis.

Objective O1 — Design a clustering-enhanced hybrid ensemble model for accurate workload prediction in cloud environments. Achieved. The proposed clustering-enhanced hybrid ensemble model integrates K-means clustering with Prophet, LSTM, and Linear Regression using inverse-error-based dynamic weighting. Experimental

results presented in Chapter 3 demonstrate that the proposed model consistently achieves lower prediction errors than the individual forecasting models across both the NASA and ClarkNet datasets, confirming its effectiveness in handling heterogeneous workload patterns.

Objective O2 — Develop a MAPE-K-based proactive auto-scaling framework using multi-step-ahead workload prediction. Achieved. The proposed proactive auto-scaling framework integrates workload prediction with the MAPE-K control loop to make scaling decisions before workload demand materializes. The framework incorporates workload smoothing, prediction blending, safety margins, cooldown periods, and stability controls, resulting in improved scaling decisions while reducing unnecessary scaling actions.

Objective O3 — Implement and deploy a real containerized testbed for end-to-end evaluation. Achieved. A real-world experimental testbed was developed using Docker Swarm, Prometheus, cAdvisor, and Locust to support trace-driven workload generation and performance monitoring. The testbed provides a reproducible environment for evaluating different auto-scaling strategies under realistic cloud workloads.

Objective O4 — Evaluate and compare different provisioning strategies under diverse workload conditions. Achieved. Four provisioning strategies, namely no scaling, static provisioning, reactive auto-scaling, and proactive auto-scaling, were evaluated using both real and synthetic workload patterns. The experimental results demonstrate that the proposed proactive approach consistently improves SLA compliance, reduces request rejection and response latency, maintains stable resource utilization, and achieves these improvements with only a marginal increase in resource provisioning compared with reactive auto-scaling.

7.4 Future Research Directions

While the proposed approach shows encouraging results, there are several directions in which this work can be further improved and extended.

One important direction is extending the framework to handle multi-service microservice architectures, where applications are composed of several interdependent services. In such environments, scaling decisions for one service often impact others. Capturing these dependencies can lead to more coordinated scaling, helping to avoid bottlenecks and improving overall system efficiency.

Another limitation of the current work is its focus on CPU-bound microservices. In real-world scenarios, many applications are I/O-bound or memory-intensive, and their behavior differs significantly from CPU-heavy workloads. Exploring scaling strategies tailored to these types of services, as well as mixed workloads, can make the system more adaptable and practical across diverse cloud applications.

In addition, combining vertical auto-scaling (adjusting CPU and memory limits) with horizontal scaling (changing the number of replicas) presents a promising direction. Such a hybrid strategy can offer more precise control over resource allocation, leading to better utilization and reduced operational costs.

Another valuable extension is the integration of reactive and proactive auto-scaling into a unified hybrid strategy. While reactive methods respond to real-time system metrics, proactive approaches anticipate future workload trends using prediction models. Combining both can help achieve a better balance between responsiveness and foresight, improving SLA adherence while avoiding unnecessary resource provisioning.

Finally, reinforcement learning (RL) can be explored for improving scaling policies. Since auto-scaling is inherently a sequential decision-making problem, RL-based approaches can learn to make smarter decisions over time by balancing performance requirements and cost under varying workload conditions.

7.5 Concluding Remarks

This thesis has shown that proactive, prediction-driven auto-scaling, when combined with carefully designed stability controls, offers a practical and effective solution to the limitations of traditional reactive systems. The proposed hybrid ensemble predictor and

stability-enhanced decision model together achieve a strong balance between service quality and resource efficiency.

The findings validate the central idea of this research: by looking ahead instead of only reacting, we can build more responsive, reliable, and efficient cloud systems. It is hoped that the framework, methods, and insights presented in this thesis will serve as a useful foundation for future advancements in autonomous cloud resource management.

References

[1] Deloitte. *Milliseconds Make Millions: Why Mobile Speed Can Slow or Grow Your Business*. Tech. rep. Deloitte, 2020. URL: https://www2.deloitte.com/content/dam/Deloitte/ie/Documents/Consulting/Milliseconds_Make_Millions_report.pdf.

[2] Philip Blunt. *How Vodafone Achieved 8% increase in sales with an instant experience*. 2021. URL: https://web.dev/vodafone/ (visited on 01/01/2024).

[3] Google. *Find Out How You Stack Up to New Industry Benchmarks for Mobile Page Speed*. 2018. URL: https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/ (visited on 01/01/2024).

[4] Peter Mell and Timothy Grance. "The NIST Definition of Cloud Computing". In: *NIST Special Publication* 800.145 (2011), pp. 1–7. DOI: 10.6028/NIST.SP.800-145.

[5] Yahya Al-Dhuraibi et al. "Elasticity in Cloud Computing: State of the Art and Research Challenges". en. In: *IEEE Transactions on Services Computing* 11.2 (Mar. 2018), pp. 430–447. ISSN: 1939-1374. DOI: 10.1109/TSC.2017.2711009.

[6] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. "Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey". In: *ACM Computing Surveys* 51.4 (2018), 73:1–73:33. DOI: 10.1145/3148149.

[7] Tao Chen, Rami Bahsoon, and Xin Yao. "A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems". In: *ACM Computing Surveys* 51.3 (2018), pp. 1–40. DOI: 10.1145/3190507.

[8] Andre Abrantes D. P. Souza and Marco A. S. Netto. "Using Application Data for SLA-Aware Auto-scaling in Cloud Environments". In: *Proceedings of the 23rd IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2015, pp. 252–255. DOI: 10.1109/MASCOTS.2015.15.

[9] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux Journal* 2014.239 (2014), p. 2.

[10] Roberto Morabito, Jimmy Kjällman, and Miika Komu. "Hypervisors vs. Lightweight Virtualization: A Performance Comparison". In: *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*. 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[11] Brendan Burns et al. "Borg, Omega, and Kubernetes". In: *ACM Queue* 14.1 (2016), pp. 70–93. DOI: 10.1145/2898442.2898444.

[12] Jeffrey O. Kephart and David M. Chess. "The Vision of Autonomic Computing". In: *IEEE Computer* 36.1 (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055.

[13] Michael Armbrust et al. "A View of Cloud Computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58. DOI: 10.1145/1721654.1721672.

[14] Rajkumar Buyya et al. "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility". In: *Future Generation Computer Systems* 25.6 (June 2009), pp. 599–616. DOI: 10.1016/j.future.2008.12.001. URL: https://doi.org/10.1016/j.future.2008.12.001.

[15] Paul Barham et al. "Xen and the art of virtualization". In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 164–177. DOI: 10.1145/1165389.945462.

[16] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting". In: *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*. 2011, pp. 500–507. DOI: 10.1109/CLOUD.2011.42.

[17] Sadeka Islam et al. "Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud". In: *Future Generation Computer Systems* 28.1 (2012), pp. 155–162. DOI: 10.1016/j.future.2011.05.027.

[18] Byoung Kim et al. "Design and Implementation of Cloud Docker Application Architecture Based on Machine Learning in Container Management for Smart Manufacturing". In: *Applied Sciences* 12 (July 2022), p. 6737. DOI: 10.3390/app12136737.

[19] Fan Zhang et al. "Quantifying cloud elasticity with container-based autoscaling". In: *Future Generation Computer Systems* 98 (2019), pp. 672–681. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2018.09.009.

[20] Thanh-Tung Nguyen et al. "Horizontal Pod Autoscaler in Kubernetes for Elastic Container Orchestration". In: *Sensors* 20.16 (2020), p. 4621. DOI: 10.3390/s20164621.

[21] N. Marie-Magdelaine, T. Ahmed, and G. Astruc-Amato. “Proactive Autoscaling for Cloud-Native Applications Using Machine Learning”. In: *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*. 2020, pp. 1–6. DOI: 10.1109/GLOBECOM42002.2020.9322147.

[22] Docker Inc. *Docker Swarm Mode Overview*. https://docs.docker.com/engine/swarm/. Accessed: 2024-09-01. 2024.

[23] Muthakshi S, K. Rajalakshmi, and Helen Josephine V.L. “Comparative analysis on containers based on kubernetes, Docker swarm, open shift and Mesos”. In: *2025 11th International Conference on Communication and Signal Processing (ICCSP)*. 2025, pp. 1398–1402. DOI: 10.1109/ICCSP64183.2025.11088783.

[24] Junjie Peng et al. “Modeling for CPU-Intensive Applications in Cloud Computing”. In: *2015 IEEE 17th International Conference on High Performance Computing and Communications*. 2015, pp. 20–25. DOI: 10.1109/HPCC-CSS-ICESS.2015.128.

[25] Junjie Peng et al. “Research on processing strategy for CPU-intensive application”. In: *Journal of Systems Architecture* 70 (2016), pp. 39–47. DOI: 10.1016/j.sysarc.2016.02.001.

[26] Emiliano Casalicchio. “Autonomic Orchestration of Containers: Problem Definition and Research Challenges”. en. In: *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools*. Taormina, Italy: ACM, 2017. ISBN: 978-1-63190-141-6. DOI: 10.4108/eai.25-10-2016.2266649. (Visited on 04/18/2026).

[27] Xiaolong Liu et al. “Automatic cloud service testing and bottleneck detection system with scaling recommendation”. In: *Concurrency and Computation: Practice and Experience* 32.1 (2020). e5161 CPE-18-0440.R2, e5161. DOI: https://doi.org/10.1002/cpe.5161.

[28] Chunlin Li, Jianhang Tang, and Youlong Luo. “Elastic edge cloud resource management based on horizontal and vertical scaling”. In: *The Journal of Supercomputing* 76.10 (2020), pp. 7707–7732. DOI: 10.1007/s11227-020-03192-3.

[29] Ali Shahidinejad, Mostafa Ghobaei-Arani, and Leila Esmaeili. “An elastic controller using colored petri nets in cloud computing environment”. In: *Cluster Computing* 23.2 (2020), pp. 1045–1071. DOI: 10.1007/s10586-019-02972-8.

[30] Jatinder Sahni and Dinesh P. Vidyarthi. “Heterogeneity aware elastic scaling of streaming applications on cloud platforms”. In: *The Journal of Supercomputing* 77.9 (2021), pp. 10512–10539. DOI: 10.1007/s11227-021-03692-w.

[31] Adel N. Toosi et al. “ElasticSFC: Auto-scaling techniques for elastic service function chaining in network functions virtualization-based clouds”. In: *Journal of Systems and Software* 152 (2019), pp. 108–119. DOI: 10.1016/j.jss.2019.02.052.

[32] J. Lee, S. Park, and H. Kim. “Proactive Auto-Scaling for Service Function Chains in Cloud Computing Based on Deep Learning”. In: *IEEE Access* 12 (2024), pp. 39771–39783. DOI: 10.1109/ACCESS.2024.3375772.

[33] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. “Elasticity in Cloud Computing: What It Is, and What It Is Not”. In: (2013), pp. 23–27. DOI: 10.1184/R1/6609497.v1.

[34] Tania Lorido-Botrán, Jose Miguel-Alonso, and Jose Antonio Lozano. “A Review of Auto-Scaling Techniques for Elastic Applications in Cloud Environments”. In: *Journal of Grid Computing* 12.4 (2014), pp. 559–592. DOI: 10.1007/s10723-014-9314-7.

[35] Yahya Al-Dhuraibi et al. “Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER”. en. In: *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. Honolulu, CA, USA: IEEE, June 2017, pp. 472–479. ISBN: 978-1-5386-1993-3. DOI: 10.1109/CLOUD.2017.67.

[36] Yahya Al-Dhuraibi et al. “Coordinating Vertical Elasticity of both Containers and Virtual Machines”. In: *Proceedings of the 8th International Conference on Cloud Computing and Services Science - CLOSER*. INSTICC. SciTePress, 2018, pp. 322–329. ISBN: 978-989-758-295-0. DOI: 10.5220/0006652403220329.

[37] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. “Elastic Virtual Machine for Fine-Grained Cloud Resource Provisioning”. en. In: *Global Trends in Computing and Communication Systems*. Ed. by P. Venkata Krishna, M. Rajasekhara Babu, and Ezendu Ariwa. Vol. 269. Series Title: Communications in Computer and Information Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 11–25. ISBN: 978-3-642-29218-7 978-3-642-29219-4. DOI: 10.1007/978-3-642-29219-4_2. (Visited on 04/18/2026).

[38] André Bauer et al. “Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field”. In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (2019), pp. 800–813. DOI: 10.1109/TPDS.2018.2870389.

[39] Dinh-Dai Vu, Minh-Ngoc Tran, and Younghan Kim. “Predictive Hybrid Autoscaling for Containerized Applications”. In: *IEEE Access* 10 (2022), pp. 109768–109778. DOI: 10.1109/ACCESS.2022.3214985.

[40] A. K. Mogal and V. P. Sonaje. “Predictive Autoscaling for Containerized Applications Using Machine Learning”. In: *Proceedings of the International Conference on Computing, Communication and Green Engineering (IC-CGU)*. 2024, pp. 1–6. DOI: 10.1109/ic-cgu58078.2024.10530773.

[41] Anna Pavlenko et al. “Vertically Autoscaling Monolithic Applications with CaaSPER: Scalable Container-as-a-Service Performance Enhanced Resizing Algorithm for the Cloud”. In: *Companion of the 2024 International Conference on Management of Data*. SIGMOD ’24. Santiago, Chile: Association for Computing Machinery, 2024, pp. 241–254. ISBN: 9798400704222. DOI: 10.1145/3626246.3653378.

[42] Byeonghui Jeong and Young-Sik Jeong. “Autoscaling techniques in cloud-native computing: A comprehensive survey”. en. In: *Computer Science Review* 58 (Nov. 2025), p. 100791. ISSN: 15740137. DOI: 10.1016/j.cosrev.2025.100791. URL: https://linkinghub.elsevier.com/retrieve/pii/S157401372500067X (visited on 04/18/2026).

[43] Vladimir Podolskiy, Anshul Jindal, and Michael Gerndt. “IaaS Reactive Autoscaling Performance Challenges”. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 954–957. DOI: 10.1109/CLOUD.2018.00144.

[44] Shutian Luo et al. “The power of prediction: microservice auto scaling via workload learning”. en. In: *Proceedings of the 13th Symposium on Cloud Computing*. San Francisco California: ACM, Nov. 2022, pp. 355–369. ISBN: 978-1-4503-9414-7. DOI: 10.1145/3542929.3563477. (Visited on 04/18/2026).

[45] Martin Straesser et al. “Why Is It Not Solved Yet? Challenges for Production-Ready Autoscaling”. In: *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. ICPE ’22. Beijing, China: Association for Computing Machinery, 2022, pp. 105–115. ISBN: 9781450391436. DOI: 10.1145/3489525.3511680.

[46] Maria Carla Calzarossa, Luisa Massari, and Daniele Tessera. “Workloads in the Clouds: An Overview”. In: *Workload Characterization: Performance Evaluation and Benchmarking*. Vol. 8962. Lecture Notes in Computer Science. Springer, 2014, pp. 1–15. DOI: 10.1007/978-3-319-30599-8_20.

[47] J. D. Freitas et al. “Evaluating Window Size Effects on Univariate Time Series Forecasting with Machine Learning”. In: *Journal of Information and Data Management* 16.1 (2025), pp. 213–223. DOI: 10.5753/jidm.2025.4668.

[48] Deepika Saxena et al. “Performance Analysis of Machine Learning Centered Workload Prediction Models for Cloud”. In: *IEEE Transactions on Parallel and Distributed Systems* 34.4 (2023), pp. 1313–1330. DOI: 10.1109/TPDS.2023.3240567.

[49] Thanh-Tung Nguyen et al. “Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration”. en. In: *Sensors* 20.16 (Aug. 2020), p. 4621. ISSN: 1424-8220. DOI: 10.3390/s20164621.

[50] S. Taherizadeh and M. Grobelnik. “Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications”. In: *Advances in Engineering Software* 140 (2020), p. 102734. DOI: 10.1016/j.advengsoft.2019.102734.

[51] Tarek Menouer, Christophe Cérin, and Patrice Darmon. “Reactive Autoscaling of Kubernetes Nodes”. In: *Proceedings of the 4th Workshop on Flexible Resource and Application Management on the Edge*. FRAME ’24. Pisa, Italy: Association for Computing Machinery, 2024, pp. 31–38. ISBN: 9798400706417. DOI: 10.1145/3659994.3660310.

[52] S. N. Srirama, M. Adhikari, and S. Paul. “Application deployment using containers with auto-scaling for microservices in cloud environment”. In: *Journal of Network and Computer Applications* 160 (2020), p. 102629. DOI: 10.1016/j.jnca.2020.102629.

[53] Gerta Sheganaku et al. “Cost-efficient auto-scaling of container-based elastic processes”. In: *Future Generation Computer Systems* 138 (2023), pp. 296–312. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2022.09.001.

[54] Fan-Hsun Tseng et al. “A Lightweight Autoscaling Mechanism for Fog Computing in Industrial Applications”. In: *IEEE Transactions on Industrial Informatics* 14.10 (2018), pp. 4529–4537. DOI: 10.1109/TII.2018.2799230.

[55] EunChan Park, KyeongDeok Baek, and In-Young Ko. “HiDRA: A Hierarchical Decentralized Reactive Autoscaler for Spatio-temporal Bursts of Load”. In: *2024 IEEE International Conference on Web Services (ICWS)*. 2024, pp. 1347–1349. DOI: 10.1109/ICWS62655.2024.00165.

[56] Hamid Arabnejad et al. “An Auto-Scaling Cloud Controller Using Fuzzy Q-Learning—Implementation in OpenStack”. In: *Proceedings of the European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Vol. 9846. LNCS. 2016, pp. 152–167. DOI: 10.1007/978-3-319-44482-6_10.

[57] Joao Paulo Karol Santos Nunes et al. “Self-adaptive, Requirements-driven Autoscaling of Microservices”. In: *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS ’24. Lisbon, AA, Portugal: Association for Computing Machinery, 2024, pp. 168–174. ISBN: 9798400705854. DOI: 10.1145/3643915.3644094.

[58] Suhrid Gupta, Muhammed Tawfiqul Islam, and Rajkumar Buyya. *A Hybrid Reactive-Proactive Auto-scaling Algorithm for SLA-Constrained Edge Computing*. 2025. arXiv: 2512.14290 [cs.DC].

[59] George E. P. Box and Gwilym M. Jenkins. *Time Series Analysis: Forecasting and Control*. San Francisco, CA: Holden-Day, 1970.

[60] R. S. Shariffdeen et al. “Workload and Resource Aware Proactive Auto-Scaler for PaaS Cloud”. In: *Proceedings of the IEEE 9th International Conference on Cloud Computing (CLOUD)*. 2016, pp. 11–18. DOI: 10.1109/CLOUD.2016.0012.

[61] Rodrigo N. Calheiros et al. “Workload Prediction Using ARIMA Model and Its Impact on Cloud Applications’ QoS”. In: *IEEE Transactions on Cloud Computing* 3.4 (2015), pp. 449–458. DOI: 10.1109/TCC.2014.2350475.

[62] Waheed Iqbal, Abdelkarim Erradi, and Arif Mahmood. “Dynamic Workload Patterns Prediction for Proactive Auto-Scaling of Web Applications”. In: *Journal of Network and Computer Applications* 124 (2018), pp. 94–107. DOI: 10.1016/j.jnca.2018.09.023.

[63] L. Al Qassem, T. Stouraitis, and E. Damiani. “Proactive Random-Forest Autoscaler for Microservice Resource Allocation”. In: *IEEE Access* 11 (2023), pp. 5944–5955. DOI: 10.1109/ACCESS.2023.3234021.

[64] Mohammad Abdullah, Waheed Iqbal, and Abdelkarim Erradi. “Predictive Autoscaling of Microservices Hosted in Fog Microdata Center”. In: *IEEE Access* 9 (2021), pp. 112916–112930. DOI: 10.1109/ACCESS.2021.3103895.

[65] Subrota Kumar Mondal et al. “Toward Optimal Load Prediction and Customizable Autoscaling Scheme for Kubernetes”. In: *Mathematics* 11.12 (2023), p. 2675. DOI: 10.3390/math11122675.

[66] Ehsan Golshani and Mehrdad Ashtiani. “Proactive auto-scaling for cloud environments using temporal convolutional neural networks”. In: *Journal of Parallel and Distributed Computing* 154 (2021), pp. 119–141. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2021.04.006.

[67] Chunyang Meng et al. “DeepScaler: Holistic Autoscaling for Microservices Based on Spatiotemporal GNN with Adaptive Graph Learning”. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2023, pp. 53–65. DOI: 10.1109/ASE56229.2023.00038.

[68] Bablu Kumar et al. “Optimizing resource allocation using proactive scaling with predictive models and custom resources”. In: *Computers and Electrical Engineering* 118 (2024), p. 109419. DOI: 10.1016/j.compeleceng.2024.109419.

[69] Mohammad Bany Taha et al. “Proactive Auto-Scaling for Service Function Chains in Cloud Computing Based on Deep Learning”. In: *IEEE Access* 12 (2024), pp. 38575–38593. DOI: 10.1109/ACCESS.2024.3375772.

[70] Tejas Subramanya and Roberto Riggio. “Centralized and Federated Learning for Predictive VNF Autoscaling in Multi-Domain 5G Networks and Beyond”. In: *IEEE Transactions on Network and Service Management* 18.1 (2021), pp. 63–78. DOI: 10.1109/TNSM.2021.3050955.

[71] Ziliang Wang et al. “DeepScaling: Autoscaling Microservices With Stable CPU Utilization for Large Scale Production Cloud Systems”. In: *IEEE/ACM Transactions on Networking* 32.5 (2024), pp. 3961–3976. DOI: 10.1109/TNET.2024.3400953.

[72] Tao Shi et al. “Auto-Scaling Containerized Applications in Geo-Distributed Clouds”. In: *IEEE Transactions on Services Computing* 16.6 (2023), pp. 4261–4274. DOI: 10.1109/TSC.2023.3317262.

[73] Fang Peng et al. “PPO Based Reinforcement Learning Horizontal Pods Auto-Scaling-A Case Study in Simulation Environment”. In: *2023 International Conference on Advances in Electrical Engineering and Computer Applications (AEECA)*. 2023, pp. 887–893. DOI: 10.1109/AEECA59734.2023.00162.

[74] Haoran Qiu et al. “AWARE: Automate Workload Autoscaling with Reinforcement Learning in Production Cloud Systems”. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2023, pp. 387–402. URL: https://www.usenix.org/conference/atc23/presentation/qiu-haoran.

[75] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[76] Mhamed Imdoukh, Imtiaz Ahmad, and Mohammad G. Alfailakawi. "Machine Learning-Based Auto-Scaling for Containerized Applications". In: *Neural Computing and Applications* 32.13 (2020), pp. 9745–9760. DOI: 10.1007/s00521-019-04507-z.

[77] Nhat-Minh Dang-Quang and Myungsik Yoo. "Deep Learning-Based Autoscaling Using Bidirectional Long Short-Term Memory for Kubernetes". In: *Applied Sciences* 11.9 (2021). ISSN: 2076-3417. DOI: 10.3390/app11093835.

[78] Nhat-Minh Dang-Quang and Myungsik Yoo. "HANSEL: Adaptive Horizontal Scaling of Microservices Using Bi-LSTM". In: *Applied Soft Computing* 105 (2021), p. 107216. DOI: 10.1016/j.asoc.2021.107216.

[79] Hoa X. Nguyen, Shaoshu Zhu, and Mingming Liu. "Graph-PHPA: Graph-based Proactive Horizontal Pod Autoscaling for Microservices using LSTM-GNN". In: *2022 IEEE 11th International Conference on Cloud Networking (CloudNet)*. 2022, pp. 237–241. DOI: 10.1109/CloudNet55617.2022.9978781.

[80] S. Luo et al. "The power of prediction: Microservice auto scaling via workload learning". In: *Proceedings of the 13th Symposium on Cloud Computing*. 2022, pp. 355–369. DOI: 10.1145/3542929.3563477.

[81] Ashish Vaswani et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 30. 2017. DOI: 10.48550/arXiv.1706.03762.

[82] Bablu Kumar et al. "Optimizing resource allocation in cloud-native applications through proactive autoscaling with the InformerAutoScale model". en. In: *The Journal of Supercomputing* 81.9 (June 2025), p. 1077. ISSN: 1573-0484. DOI: 10.1007/s11227-025-07500-7. URL: https://link.springer.com/10.1007/s11227-025-07500-7 (visited on 04/18/2026).

[83] Bablu Kumar, Anshul Verma, and Pradeepika Verma. "A multivariate transformer-based monitor-analyze-plan-execute (MAPE) autoscaling framework for dynamic resource allocation in cloud environment". en. In: *Computing* 107.3 (Mar. 2025), p. 69. ISSN: 0010-485X, 1436-5057. DOI: 10.1007/s00607-025-01426-x. (Visited on 04/18/2026).

[84] Mouhamadou Bâ, Chandra Harsha, and Hari Subramanian. "Leveraging Interpretability in the Transformer to Automate Proactive Scaling of Cloud Resources". In: *IEEE Transactions on Cloud Computing* 12.4 (2024), pp. 1043–1057. DOI: 10.48550/arXiv.2409.03103.

[85] Ding Zou et al. "OptScaler: A Collaborative Framework for Robust Autoscaling in the Cloud". In: *Proc. VLDB Endow.* 17.12 (Aug. 2024), pp. 4090–4103. ISSN: 2150-8097. DOI: 10.14778/3685800.3685829.

[86] Kaan Aykurt et al. “HyPA: Hybrid Horizontal Pod Autoscaling with Automated Model Updates”. In: *2023 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2023, pp. 8–14. DOI: 10.1109/NFV-SDN59219.2023.10329742.

[87] Víctor Rampérez et al. “FLAS: A Combination of Proactive and Reactive Auto-Scaling Architecture for Distributed Services”. In: *Future Generation Computer Systems* 117 (2021), pp. 422–436. DOI: 10.1016/j.future.2020.12.025.

[88] A. Lipari, G. P. Mattia, and R. Beraldi. “Dynamic and Forecast-Based Containers Autoscaling for Kubernetes with Reinforcement Learning”. In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2025, pp. 1081–1088. DOI: 10.1109/IPDPSW66978.2025.00169.

[89] Chamath Wanigasooriya and Indrajith Ekanayake. “NimbusGuard: A Novel Framework for Proactive Kubernetes Autoscaling Using Deep Q-Networks”. In: *Proceedings of the 40th International Conference on Information Networking (ICOIN)*. 2026, pp. 726–731. DOI: 10.1109/ICOIN68469.2026.11480646.

[90] Pratik Mishra et al. “Optimizing Cloud Workloads: Autoscaling with Reinforcement Learning”. In: *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*. 2024, pp. 217–222. DOI: 10.1109/CLOUD62652.2024.00033.

[91] Huaijun Wang et al. “Container Scaling Strategy Based on Reinforcement Learning”. In: *Security and Communication Networks* 2023.1 (2023), p. 7400235. DOI: https://doi.org/10.1155/2023/7400235.

[92] Muhammad Abdullah et al. “Burst-Aware Predictive Autoscaling for Containerized Microservices”. In: *IEEE Transactions on Services Computing* 15.3 (2022), pp. 1448–1460. DOI: 10.1109/TSC.2020.2995937.

[93] Thomas G. Dietterich. “Ensemble Methods in Machine Learning”. In: *Multiple Classifier Systems*. Ed. by Josef Kittler and Fabio Roli. Vol. 1857. Lecture Notes in Computer Science. Springer, 2000, pp. 1–15. DOI: 10.1007/3-540-45014-9_1.

[94] Hoang Minh Nguyen et al. “ESNemble: An Echo State Network-Based Ensemble for Workload Prediction and Resource Allocation of Web Applications in the Cloud”. In: *The Journal of Supercomputing* 75 (2019), pp. 6303–6323. DOI: 10.1007/s11227-019-02851-4.

[95] Parminder Singh, Pooja Gupta, and Kiran Jyoti. "TASM: Technocrat ARIMA and SVR Model for Workload Prediction of Web Applications in Cloud". In: *Cluster Computing* 22.2 (2019), pp. 619–633. DOI: 10.1007/s10586-018-2868-6.

[96] P. Nikhil and G. Kumar. "Forecasting Cloud Application Workloads with Cloud Insight for Predictive Resource Management". In: *International Scientific Journal of Engineering and Management* 7.2 (2025), pp. 1–8. DOI: 10.55041/ISJEM04854.

[97] L. Toka et al. "Machine Learning-Based Scaling Management for Kubernetes Edge Clusters". In: *IEEE Transactions on Network and Service Management* 18.1 (2021), pp. 958–972. DOI: 10.1109/TNSM.2021.3052837.

[98] S. Thiruchadai Pandeeswari et al. "Predictive Cloud Resource Management by Workload Forecasting with an Ensemble of Hybrid ARIMA-LSTM and EL Models". In: *Anadolu Psikiyatri Dergisi* 26.2 (2025). DOI: 10.5281/zenodo.15038432.

[99] P. B. Guruge and Y. H. P. P. Priyadarshana. "Time series forecasting-based Kubernetes autoscaling using Facebook Prophet and Long Short-Term Memory". In: *Frontiers in Computer Science* 7 (2025), p. 1509165. DOI: 10.3389/fcomp.2025.1509165. URL: https://doi.org/10.3389/fcomp.2025.1509165.

[100] James B. MacQueen. "Some Methods for Classification and Analysis of Multivariate Observations". In: *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*. Vol. 1. 1967, pp. 281–297.

[101] B. Feng and Z. Ding. "Application-Oriented Cloud Workload Prediction: A Survey and New Perspectives". In: *Tsinghua Science and Technology* 30.1 (2025), pp. 34–54. DOI: 10.26599/TST.2024.9010024.

[102] Jitendra Kumar and Ashutosh Kumar Singh. "Workload prediction in cloud using artificial neural network and adaptive differential evolution". en. In: *Future Generation Computer Systems* 81 (Apr. 2018), pp. 41–52. ISSN: 0167739X. DOI: 10.1016/j.future.2017.10.047. (Visited on 04/18/2026).

[103] NASA Kennedy Space Center. *NASA-HTTP: HTTP Requests to the NASA Kennedy Space Center WWW Server*. http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html. Accessed: 2024-01-15. 1995.

[104] ClarkNet Internet Service Provider. *ClarkNet-HTTP: HTTP Requests to ClarkNet WWW Server*. http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html. Accessed: 2024-01-15. 1995.

[105] Sean J. Taylor and Benjamin Letham. “Forecasting at Scale”. In: *The American Statistician* 72.1 (2018), pp. 37–45. DOI: 10.1080/00031305.2017.1380080.

[106] Y. Bengio, P. Simard, and P. Frasconi. “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1109/72.279181.

[107] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *arXiv preprint arXiv:1406.1078* (2014). DOI: 10.48550/arXiv.1406.1078.

[108] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. “An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling”. In: *arXiv preprint arXiv:1803.01271* (2018). URL: https://arxiv.org/abs/1803.01271.

[109] A. Mahmoud and A. Mohammed. “Leveraging Hybrid Deep Learning Models for Enhanced Multivariate Time Series Forecasting”. In: *Neural Processing Letters* 56 (2024). Accepted: 21 May 2024, Published: 23 August 2024, p. 223. DOI: 10.1007/s11063-024-11656-3.

[110] Amazon Web Services. *Amazon EC2 Auto Scaling*. https://aws.amazon.com/autoscaling/. Accessed: 2026-05-14. 2026.

[111] Stefan Frey et al. “Cloud QoS Scaling by Fuzzy Logic”. In: *2014 IEEE International Conference on Cloud Engineering*. 2014, pp. 343–348. DOI: 10.1109/IC2E.2014.30.

[112] Khaled Salah, Khalid Elbadawi, and Raouf Boutaba. “An Analytical Model for Estimating Cloud Resources of Elastic Services”. In: *Journal of Network and Systems Management* 24.2 (2016), pp. 285–308. DOI: 10.1007/s10922-015-9352-x.

[113] Pradeep Padala et al. “Automated Control of Multiple Virtualized Resources”. In: *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys ’09)*. 2009, pp. 13–26. DOI: 10.1145/1519065.1519068.

[114] Alessandro Gambi, Mauro Pezzè, and Giovanni Toffetti. “Kriging-Based Self-Adaptive Cloud Controllers”. In: *IEEE Transactions on Services Computing* 9.3 (2016), pp. 368–381. DOI: 10.1109/TSC.2015.2389236.

[115] Gerald Tesauro et al. “On the Use of Hybrid Reinforcement Learning for Autonomic Resource Allocation”. In: *Cluster Computing* 10.3 (2007), pp. 287–299. DOI: 10.1007/s10586-007-0035-6.

[116] Anshul Gandhi et al. “AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers”. In: *ACM Trans. Comput. Syst.* 30.4 (Nov. 2012). ISSN: 0734-2071. DOI: 10.1145/2382553.2382556.

[117] H. Fernandez, G. Pierre, and T. Kielmann. “Autoscaling web applications in heterogeneous cloud infrastructures”. In: *Proceedings of 2014 IEEE International Conference on Cloud Engineering (IC2E)*. 2014, pp. 195–204. DOI: 10.1109/IC2E.2014.25.

[118] Upendra Sharma et al. “A Cost-Aware Elasticity Provisioning System for the Cloud”. In: *2011 31st International Conference on Distributed Computing Systems*. 2011, pp. 559–570. DOI: 10.1109/ICDCS.2011.59.

[119] Nedeljko Vasić et al. “DejaVu: accelerating resource allocation in virtualized environments”. In: *SIGPLAN Not.* 47.4 (Mar. 2012), pp. 423–436. ISSN: 0362-1340. DOI: 10.1145/2248487.2151021.

[120] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. “Resource Provisioning of Web Applications in Heterogeneous Clouds”. In: *2nd USENIX Conference on Web Application Development (WebApps 11)*. Portland, OR: USENIX Association, June 2011. URL: https://www.usenix.org/conference/webapps11/resource-provisioning-web-applications-heterogeneous-clouds.

[121] K8s Guide. *Kubernetes Horizontal Pod Autoscaling Explained (HPA and Metrics)*. Accessed: 2026-05-20. 2026. URL: https://www.k8s.guide/workloads/scaling-hpa/.

[122] Locust Contributors. *Locust – An Open Source Load Testing Tool*. https://locust.io. Accessed: 2024-09-01. 2024.

[123] Jeffrey O. Kephart and David M. Chess. “The Vision of Autonomic Computing”. In: *Computer* 36.1 (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055.

[124] Prometheus Authors. *Prometheus – Monitoring System and Time Series Database*. https://prometheus.io. Accessed: 2024-09-01. 2024.

[125] Google LLC. *cAdvisor: Container Advisor*. https://github.com/google/cadvisor. Accessed: 2024-09-01. 2024.

[126] Jeffrey Dean and Luís André Barroso. “The Tail at Scale”. In: *Communications of the ACM* 56.2 (2013), pp. 74–80. DOI: 10.1145/2408776.2408794.

List of Publications

1. Trivedi, N. S., & Upadhyaya, A. N. (2026). ‘ "Hybrid Ensemble Approach for Accurate Workload Prediction in Dynamic Cloud Environments." SSRG International Journal of Electronics and Communication Engineering, 13(2), 180–194. https://doi.org/10.14445/23488549/IJECE-V13I2P114

2. Trivedi, N. S., & Panchal, S. D. (2023). “Optimization enabled elastic scaling in cloud based on predicted load for resource management.” Multiagent and Grid Systems, 19(4), 289–311. https://doi.org/10.3233/MGS-230003

Appendix A

Experimental Dataset Characteristics

A.1 NASA HTTP Access Log Dataset

The NASA HTTP dataset [103] records HTTP requests to the NASA Kennedy Space Center web server from 1 July to 31 August 1995, comprising approximately 3.46 million requests. The raw log is parsed to extract request timestamps and aggregated into one-minute intervals. After removing the maintenance gap between the two months, the resulting time series contains 44,640 samples spanning 31 effective days. The workload has a mean of 41.23 requests/minute, median of 32, standard deviation of 28.45, and values ranging from 0 to 248 requests/minute. It exhibits strong daily periodicity along with irregular bursts and spike events.

TABLE A.1: Statistical summary of the NASA HTTP Access Log dataset.

Property	**Value**
Collection period	1 July – 31 August 1995
Approximate requests	3.46 million
Total samples	44,640
Sampling interval	1 minute
Mean	41.23 requests/min
Median	32 requests/min
Maximum	248 requests/min
Minimum	0 requests/min
Standard deviation	28.45 requests/min

A.2 ClarkNet WWW Dataset

The ClarkNet WWW dataset [104] records HTTP requests to the ClarkNet Internet service provider from 28 August to 3 September 1995, comprising approximately 3.33

million requests. After aggregation into one-minute intervals, the dataset contains 8,899 samples over approximately seven days. The workload has a mean of 52.67 requests/minute, median of 45, standard deviation of 35.12, and values ranging from 0 to 312 requests/minute. Compared with NASA, ClarkNet exhibits more irregular short-term variations and pronounced non-linear bursts.

TABLE A.2: Statistical summary of the ClarkNet WWW dataset.

Property	**Value**
Collection period	28 August – 3 September 1995
Approximate requests	3.33 million
Total samples	8,899
Sampling interval	1 minute
Mean	52.67 requests/min
Median	45 requests/min
Maximum	312 requests/min
Minimum	0 requests/min
Standard deviation	35.12 requests/min

A.3 Synthetic Workload Traces

Two synthetic workload traces were generated to complement the real HTTP workloads during the auto-scaling experiments.

Monotonically Increasing Trace. The request rate increases linearly from 5 RPS to 100 RPS over a 60-minute experiment, representing sustained workload growth.

Periodic Trace. The request rate varies periodically between 10 RPS and 90 RPS with a 15-minute cycle, allowing evaluation of the proactive controller under repeating workload patterns.

Appendix B

Experimental Infrastructure Configuration

B.1 AWS EC2 Instance Specifications

All three nodes of the Docker Swarm cluster were provisioned as Amazon EC2 `t3.small` instances in the `us-east-1` region.

TABLE B.1: EC2 instance configuration for all cluster nodes.

Parameter	Value
Instance type	`t3.small`
vCPUs	2
RAM	4 GB
Processor	Intel Xeon E5-2686 v4
Base clock	2.3 GHz
Operating system	Ubuntu 20.04 LTS
Region	`us-east-1`

B.2 Docker Stack Deployment Configuration

This section provides the Docker Stack configuration used to deploy the experimental environment for the proposed proactive auto-scaling framework. The stack file defines the shared overlay network, local registry, cAdvisor, Prometheus, Prime application service, autoscaler service, and Locust workload generator. It also specifies placement constraints, service ports, mounted volumes, resource limits, replica bounds, cooldown period, prediction horizon, per-replica capacity parameters, and SLA-related configuration.

```
version: "3.8"
```

TABLE B.2: Software stack versions used in the experimental platform.

Component	**Version**
Docker Engine (leader)	25.0.6
Docker Engine (workers)	20.10.7
Python	3.9.7
Flask	2.1.0
Gunicorn	20.1.0
Prometheus	2.12.0
cAdvisor	0.25.0
Locust	2.14.2
TensorFlow/Keras	2.11.0
scikit-learn	1.2.0
Prophet	1.1.1
NumPy	1.23.5
Pandas	1.5.3

```
networks:
  autoscale:
    external: true
    name: autoscaler_autoscale

configs:
  prometheus_config_v2:
    file: ./prometheus/prometheus.yml

services:
  registry:
    image: registry:2
    networks: [autoscale]
    ports:
      - "5000:5000"
    deploy:
      placement:
        constraints: [node.role == manager]

  cadvisor:
```

```
    image: google/cadvisor:v0.25.0
    networks: [autoscale]
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
      - /:/rootfs:ro
      - /var/run:/var/run:ro
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
    deploy:
      mode: global

  prometheus:
    image: prom/prometheus:v2.12.0
    networks: [autoscale]
    command:
      - "--config.file=/etc/prometheus/prometheus.yml"
      - "--storage.tsdb.retention.size=1GB"
      - "--web.enable-lifecycle"
    configs:
      - source: prometheus_config_v2
        target: /etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"
    deploy:
      placement:
        constraints: [node.role == manager]

  prime_app:
    image: naimishatrivedi/prime-app:latest2
    networks: [autoscale]
    ports:
      - "8080:8000"
    deploy:
```

```
    mode: replicated
    replicas: 1
    labels:
      swarm.autoscaler: "true"
      swarm.autoscaler.minimum: "1"
      swarm.autoscaler.maximum: "12"
    resources:
      limits:
        cpus: "0.50"
        memory: 256M
      reservations:
        cpus: "0.10"
        memory: 64M

autoscaler:
  image: naimishatrivedi/proactive-autoscaler:realml17-fixed
  networks: [autoscale]
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock:rw
    - /opt/hybrid_models:/models:ro
  environment:
    - PROMETHEUS_URL=http://prometheus:9090
    - TARGET_SERVICE=oneshot_prime_app
    - MODE=proactive
    - INFER_SCRIPT=/autoscaler/predictor/infer_hybrid_real.py
  deploy:
    placement:
      constraints: [node.hostname == leader]

locust:
  image: naimishatrivedi/prime-locust:latest
  volumes:
    - /opt/autoscale/traces:/opt/autoscale/traces:ro
```

```
      - /opt/autoscale/locust:/locust
    networks: [autoscale]
    ports:
      - "8089:8089"
    deploy:
      placement:
        constraints: [node.hostname == leader]
```

LISTING B.1: Docker Stack configuration used for the experimental deployment

B.3 Flask Application Source Code (Excerpt)

The target microservice was a Flask/Gunicorn application that computed all prime numbers up to $n = 50{,}000$ per request using a trial-division algorithm with $O(n\sqrt{n})$ complexity. The per-replica throughput capacity was measured empirically as approximately 10.4 RPS under the experimental conditions.

```
from flask import Flask, jsonify
import math

app = Flask(__name__)

def compute_primes(n):
    primes = []
    for num in range(2, n + 1):
        is_prime = True
        for i in range(2, int(math.sqrt(num)) + 1):
            if num % i == 0:
                is_prime = False
                break
        if is_prime:
            primes.append(num)
    return primes

@app.route('/primes')
def primes():
    result = compute_primes(50000)
    return jsonify({'count': len(result)})
```

LISTING B.2: Flask prime-service excerpt used in experiments

B.4 Prometheus Scrape Configuration

Prometheus was configured with a 10-second global scrape interval. The following targets were scraped:

- `cAdvisor` on all three nodes (port 8080) — container-level CPU, memory, and network metrics.
- `Flask application` (port 5000) — request rate and response time histograms via the `prometheus_flask_exporter` library.
- `Node exporter` on all three nodes (port 9100) — host-level CPU and memory utilisation.