National College of Ireland

# Optimizing the Resource Utilization in Cloud Computing Environment with Autoscaling using Machine Learning Methods

MSc Research Project
Cloud Computing

SriMadhan Shettihalli Anandreddy
Student ID: x21230064

School of Computing
National College of Ireland

Supervisor: Punit Gupta

**National College of Ireland**
**Project Submission Sheet**
**School of Computing**

| | |
|---|---|
| **Student Name:** | SriMadhan Shettihalli Anandreddy |
| **Student ID:** | x21230064 |
| **Programme:** | Cloud Computing |
| **Year:** | 2023 |
| **Module:** | MSc Research Project |
| **Supervisor:** | Punit Gupta |
| **Submission Due Date:** | 18/09/2023 |
| **Project Title:** | Optimizing the Resource Utilization in Cloud Computing Environment with Autoscaling using Machine Learning Methods |
| **Word Count:** | 2031 |
| **Page Count:** | 16 |

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

**ALL** internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

| | |
|---|---|
| **Signature:** | Sri Madhan Shettihalli Anandreddy |
| **Date:** | 18th September 2023 |

**PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:**

| | |
|---|---|
| Attach a completed copy of this sheet to each project (including multiple copies). | ☐ |
| **Attach a Moodle submission receipt of the online project submission**, to each project (including multiple copies). | ☐ |
| **You must ensure that you retain a HARD COPY of the project**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer. | ☐ |

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

| **Office Use Only** | |
|---|---|
| Signature: | |
| Date: | |
| Penalty Applied (if applicable): | |

# Optimizing the Resource Utilization in Cloud Computing Environment with Autoscaling using Machine Learning Methods

SriMadhan Shettihalli Anandreddy
x21230064

## 1 Introduction

The primary goal of this document is to provide the reader with a comprehensive manual for successfully installing, configuring, and executing the project. It serves as a step-by-step guide for effectively setting up and running the system from start to finish. The document covers detailed installation instructions for each module, including relevant configuration options and parameters that may need to be set. Next, it delves into the the overall system architecture at a high level first, providing the reader with an understanding of the different components and how they fit together. With the system installed, the execution flow and workflows are described to give the reader a clear picture of how modules interact during operation. The guide Also includes troubleshooting advice for common errors and issues that may be encountered during installation or execution. Following the steps outlined in this end-to-end guide will equip the reader with the knowledge to smoothly deploy the project in a new environment.. It is highly recommended that this document be thoroughly read and understood by those looking to operate the system.

## 2 Prerequisites

Users should have a basic working knowledge of Amazon Cloud, Ubuntu, Python programming, and common machine learning algorithms before using this system. Basic proficiency in Ubuntu is required to be able to navigate the file system, install packages, and execute commands through the terminal. Users should be familiar with Ubuntu administration including managing software, users, and processes. A fundamental understanding of the Python programming language is also expected. Users should be comfortable with Python syntax, data structures, control flow, modules, and virtual environments. Additionally, users should have experience with standard machine learning algorithms and models, including concepts like training, testing, feature engineering, and performance evaluation. Exposure to libraries like Scikit-Learn for implementing machine learning in Python is beneficial.

While advanced expertise is not required, having basic Ubuntu system administration skills, working knowledge of Python programming, and grounding in core machine learning concepts and techniques will ensure users can effectively work with the system.

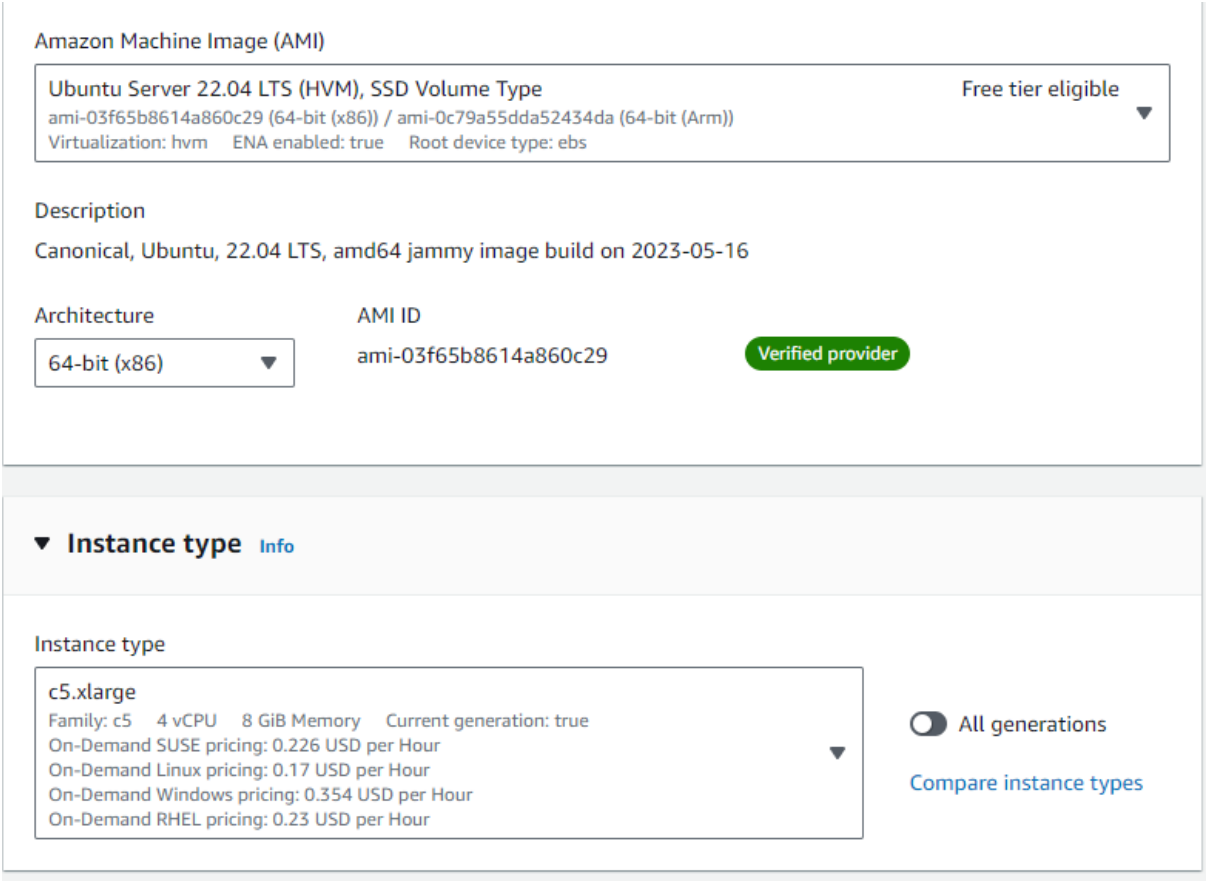


Figure 1: AWS EC2 Instance

# 3 Environment Setup

Figure 2 summarizes the hardware and software requirements for executing the project. I am using AWS cloud for my implementation. I have used configuration for EC2 instance as specified in Figure 1.

## 3.1 Packages and Libraries

I have developed my project using python programming language and python version of 3.7 or later is required. You can install python latest version from `https://www.python.org/downloads/`
We are using multiple library functions in the project, we need pip package manager to install these library functions. pip can be installed by executing the below commands as shown in Figure 3 and Figure 4

sudo apt update && sudo apt upgrade
sudo apt install python3-pip

The library functions used in the project include TensorFlow for implementing deep learning models, scikit-learn for traditional machine learning algorithms, Matplotlib for visualization, NumPy for numerical processing, and Pandas for data analysis and manipulation. Boto3 provides interfaces for AWS services like accessing s3. The code is

| Resource | Configuration |
|---|---|
| Operating System | Linux |
| Main Memory (RAM) | 4GB |
| Number of CPU Cores | 8 (Virtual Cores) |
| Storage | 30GB |
| Programming Language | Python3 |
| Python Libraries | Numpy, Pandas, Threading, Matplotlib, Sklearn, Keras, Tensorflow |

Figure 2: Hardware and software Requirements

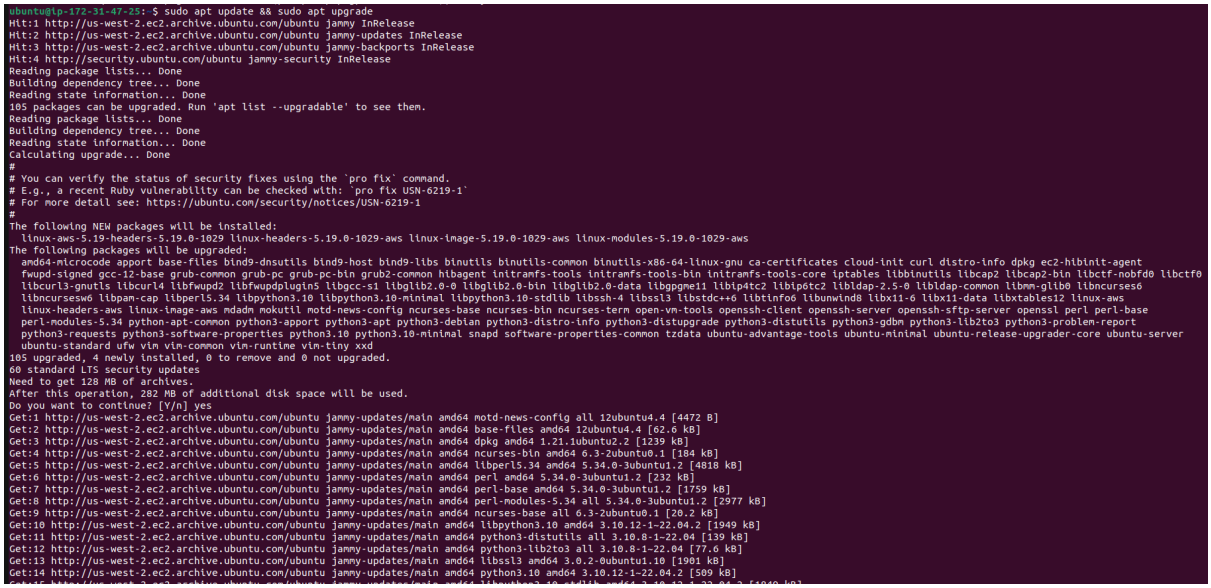


Figure 3: sudo apt update && sudo apt upgrade

developed in Python for its widespread use in machine learning and data science applications. TensorFlow enables building deep neural networks for approaches like CNNs and LSTM. Scikit-learn offers a range of classical ML algorithms like regression and random forests. Matplotlib and Pandas support analyzing and visualizing datasets. NumPy facilitates mathematical operations. Using these core libraries provides the modeling capabilities and data handling tools needed for an autoscaling prediction system based on machine learning. We can install this by executing the command as shown in Figure 5 : pip install scikit-learn tensorflow matplot

# 4 Methedology

## 4.1 Components

Project model consists of 5 main components, which are explained below.

- Data Generator: Generates simulated load data over time intervals. The load data

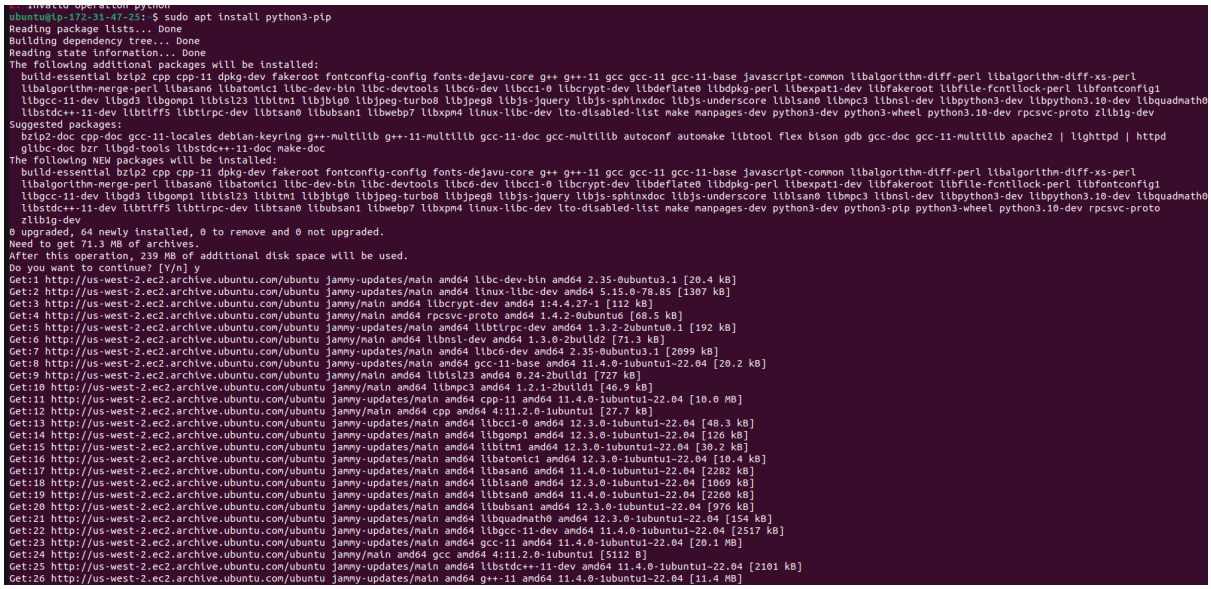


Figure 4: sudo apt install python3-pip

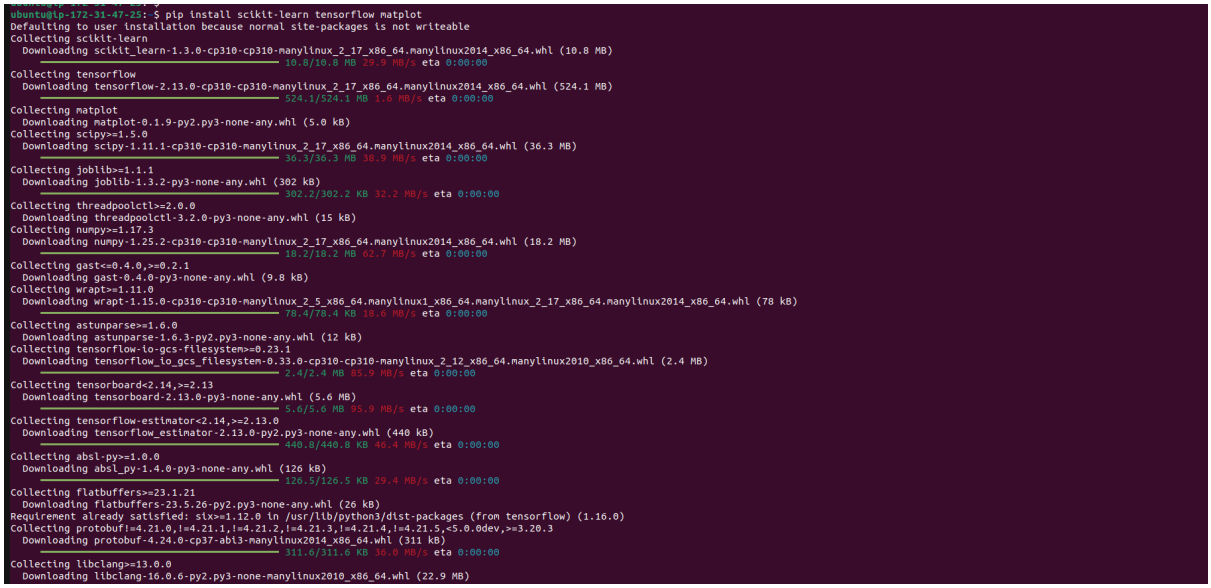


Figure 5: pip install scikit-learn tensorflow matplot

sequence.py 2

sequence.py > ...

```python
import numpy as np
import math
import random
import csv

SCALE = 1 # amplitude of the wave
VARIANCE = 0.1 # deviation from sine wave
INCREMENT = math.pi/36      #increment by this factor in each step

def generate_load_value(angle):
    N = math.sin(angle) + (random.random()*VARIANCE)*SCALE
    M = math.cos(angle)*(SCALE)
    return abs(M) + abs(N)

if __name__ == "__main__":
    angle = 0.1
    x = []
    y = []
    for i in range(180):
        angle += INCREMENT
        x.append(angle)
        y.append(generate_load_value(angle))

    # Save the data to a CSV file
    with open("generated_data.csv", "w", newline="") as csvfile:
        csvwriter = csv.writer(csvfile)
        csvwriter.writerow(["Angle", "Load Value"])
        for angle, load_value in zip(x, y):
            csvwriter.writerow([angle, load_value])
```

Figure 6: Sinusoidal Cloud workload generation

is a combination of sine and cosine wave with some randomness. Figure 6 shows code snippet for the same.

- Task Generator: This component is responsible for generating jobs. The Loader class adds generated jobs to the Scaler component and executes them. Code for handling this work is all included in jobs.py file.

- Scaler: Defines the Scaler class responsible for scaling the VM instance based on historical load data. It receives input from the Loader, maintains a history of load values, and uses a model to predict the next load value. It then adjusts the capacity of the executing node and executes the job. Code for handling this work is all included in scaler.py file.

- Resource allocation unit: this component is where actual tasks execute on. It defines the Node class that simulates a cloud instance capable of vertical scaling. The capacity of the resource allocation unit is adjusted by the Scaler. Code for handling this work is all included in node.py file. It keeps track of historical error and load data which will be utilized in comparing algorithms on the error metrics.

- Visualizer: With this component we plot graphs to visualize the data for better understanding. Code for handling this component is all written in visualizer.py file. It contains the Visualizer class responsible for generating various plots to visualize the project's performance. It creates plots related to load vs. capacity, underuse_vs_overuse, scaling error for different models, and delayed tasks for each model and stores these graphs in Amazon S3 bucket.

## 4.2 Algorithms

In this research we are using 3 different kinds of implementation for predicting auto-scaling in cloud.

### 4.2.1 Traditional threshold based method

In this method, scaling is done by a scaling factor which is predetermined by examining the historical data. Engineers analyze the load and use formula to calculate scaling factor. In our case we are calculating the scaling factor by taking the average of the historical load values. Figure 7 shows the code snippet for Rule-based model

### 4.2.2 Machine Learning Models

We are using two machine learning models for prediction of auto-scaling: Random Forest (RF) and Support Vector Regression (SVR) Model.

- Support Vector Regression: We are using a simple SVR model with linear kernel. this model works based on Simple linear boundary which is easier to interpret. It also has advantage of faster execution times when compared to Non-Linear patterns. Figure 8 shows the code snippet for SVR model.

- Random Forest: We are using simple Random forest model with shallow tree. The max_depth parameter is set to 2 and random_state is set to 0. Setting the random_state to zero ensures consistent output results. After setting this parameters

```python
class AvgModel:
    def predict(self,values):
        return sum(values)/len(values)

if __name__=="__main__":
    am = AvgModel()
    print(am.predict([1,2,3,4,5]))
```

Figure 7: Rule-based Model

```python
from sklearn.svm import SVR
import numpy as np

class SVRModel:
    def predict(self,values):
        lr = SVR(kernel="linear")
        x = np.arange(len(values),dtype=np.float32).reshape([9,1])
        y = np.array(values,dtype=np.float32)
        lr.fit(x,y)
        x_p = np.array([[len(values)]],dtype=np.float32)
        predict = lr.predict(x_p).item()
        return predict

if __name__=="__main__":
    lr = SVRModel()
    print(lr.predict([1,2,3,4,5,6,7,8,9]))
```

Figure 8: Support Vector Regression (SVR) Model

```python
from sklearn.ensemble import RandomForestRegressor
import numpy as np

class RFModel:
    def predict(self,values):
        lr = RandomForestRegressor(max_depth=2, random_state=0)
        x = np.arange(len(values),dtype=np.float32).reshape([9,1])
        y = np.array(values,dtype=np.float32)
        lr.fit(x,y)
        x_p = np.array([[len(values)]],dtype=np.float32)
        predict = lr.predict(x_p).item()
        return predict

if __name__=="__main__":
    lr = RFModel()
    print(lr.predict([1,2,3,4,5,6,7,8,9]))
```

Figure 9: Random Forest Model

we are reshaping the input values into a 2D array with one column as required by the model and train the model on these input values. Figure 9 shows the code snippet for SVR model.

#### 4.2.3 Deep Learning Models

We are predicting auto-scaling using two deep learning models: Gated recurrent unit (GRU) and an ensemble model- CNN_LSTM Model.

- CNN_LSTM Model: The primary concept behind using these models on time-series data is that while CNN models are beneficial in extracting the valuable features and may filter out the noise of the input data, LSTM networks are capable of capturing sequence pattern information. The model workflow is explained as below: Input data is loaded and reshaped to 3d array format as samples, time steps and features. A sequential model is then created and convolution model is applied for feature extraction. then LSTM is applied for sequence modelling. Finally a dense layer is added to prediction output. We are using Adam optimizer with learning rate set to 0.0001. also loss parameter is set to mean squared error. then the model is trained for 1000 epochs with batch size of 128. after 1000 epochs, the loss value remains constant. Figure 10 shows the training for CNN_LSTM model.

- Gated recurrent unit Model: the workflow of model is as follows: The input data is loaded and it contains the features x and targets y. A Sequential Keras model is created. A GRU layer is added to the model to handle the sequential nature of the data. The input shape is set to (9,1) based on the 9 time steps. A Dense output layer is added for making a single value prediction. The model is compiled with

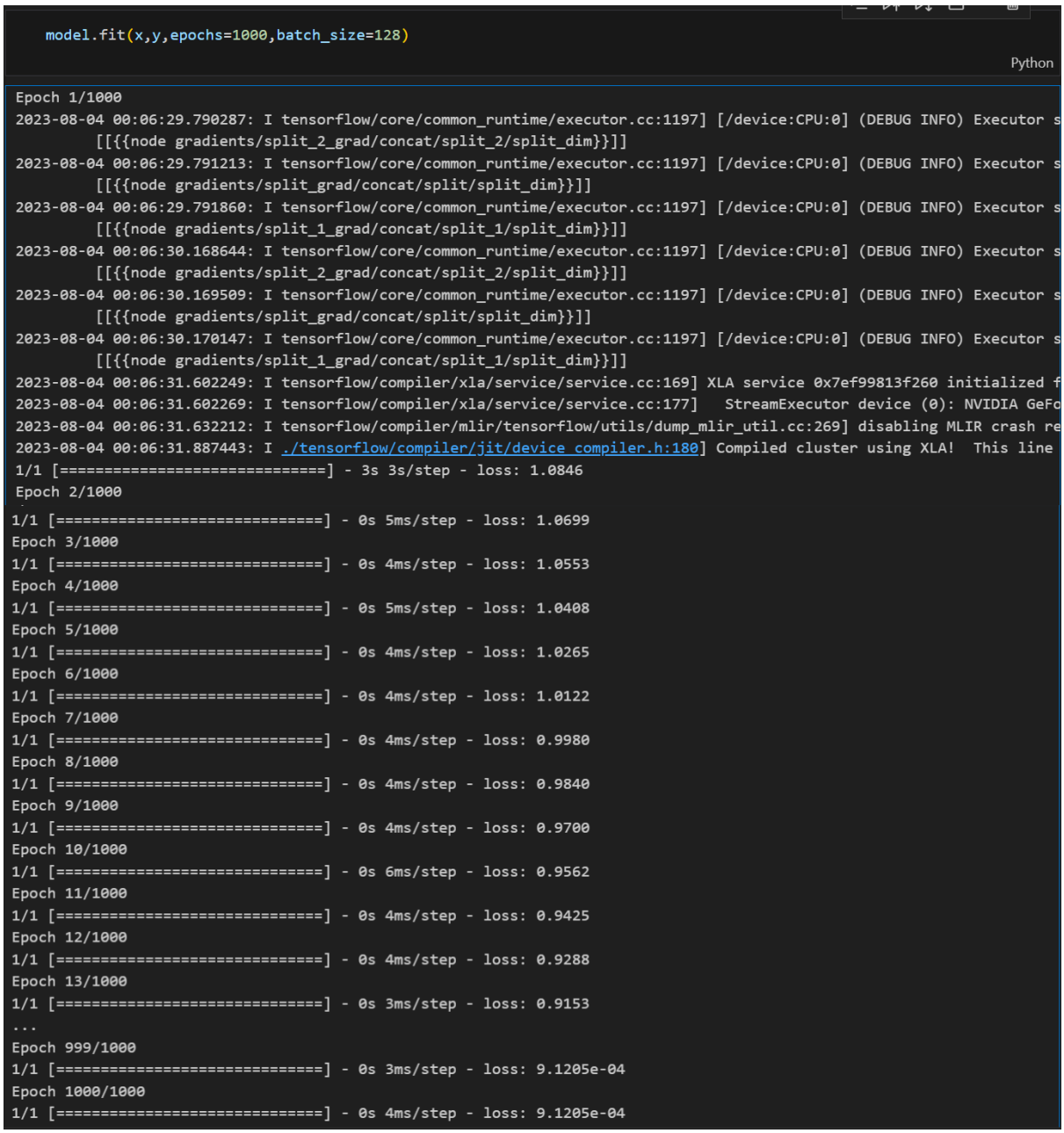


Figure 10: CNN_LSTM Model

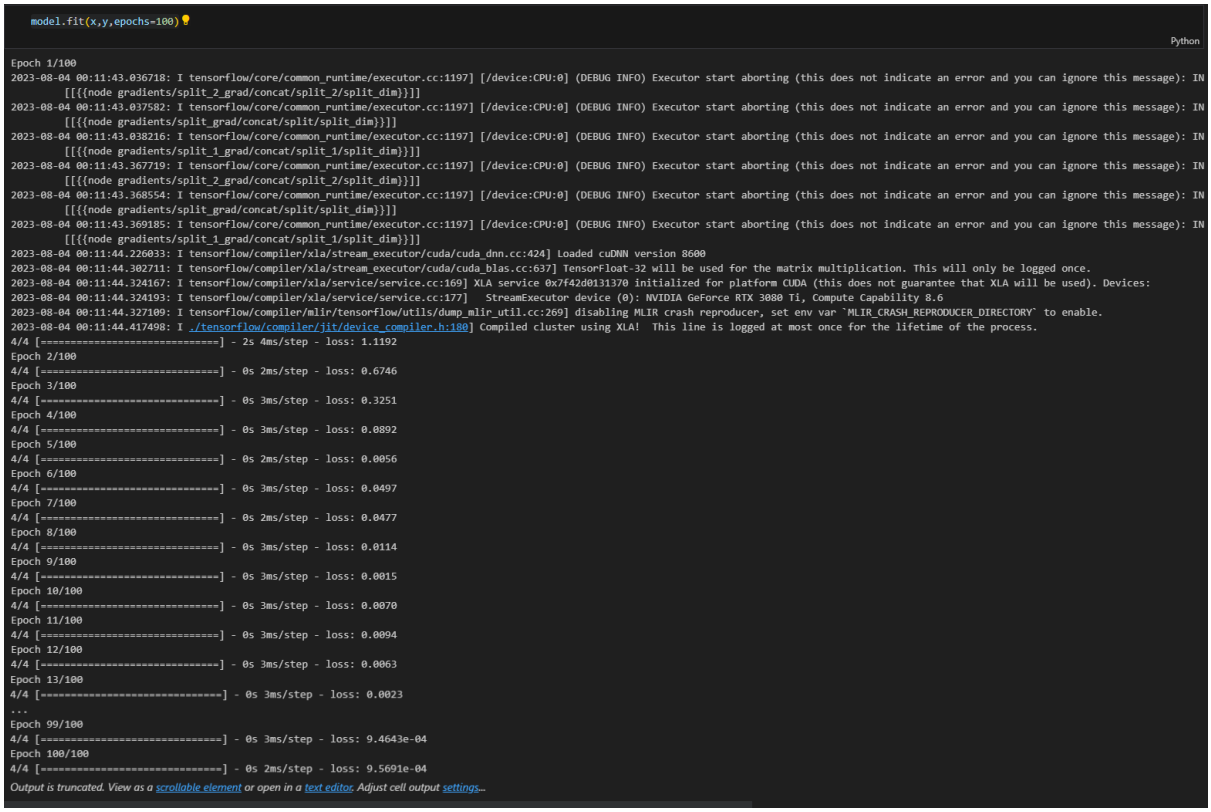


Figure 11: GRU Model

Adam optimizer using a learning rate of 0.001 and MSE loss function. The model is trained for 100 epochs by fitting on the input x and target y data. The trained GRU model is then saved. Figure 11 shows the GRU model training for 100 epochs.

# 5 Execution and Results

### 5.0.1 Execution steps

Once we have Environment setup and all the project files copied to destination path, we are now good to execute the models. We have created python files for invoking all the components specified in project methodology for each Model.

- Data generation: Code for generating data is stored in sequence.py python file. Executing this file creates ds.pkl file which will be used for training ML and deep learning algorithms. Also a sample graph is plotted to visualize the generated data. Figure 12 shows sample plot of dataset generated using sequence.py Figure 13 shows the command for generating synthetic data. We can see the output file ds.pkl created after executing the python file.

- Train Algorithms: We use the synthetic dataset generated in above step to train Moving_avg, RF, SVR, GRU, CNN_LSTM algorithms. Figure 14, Figure 15 and Figure 16 shows the output generated for Model training.
  For moving_avg we had fed input as [1,2,3,4,5] resulting in average of 3 as shown in Figure 14. We could see different predicted capacity for different models.

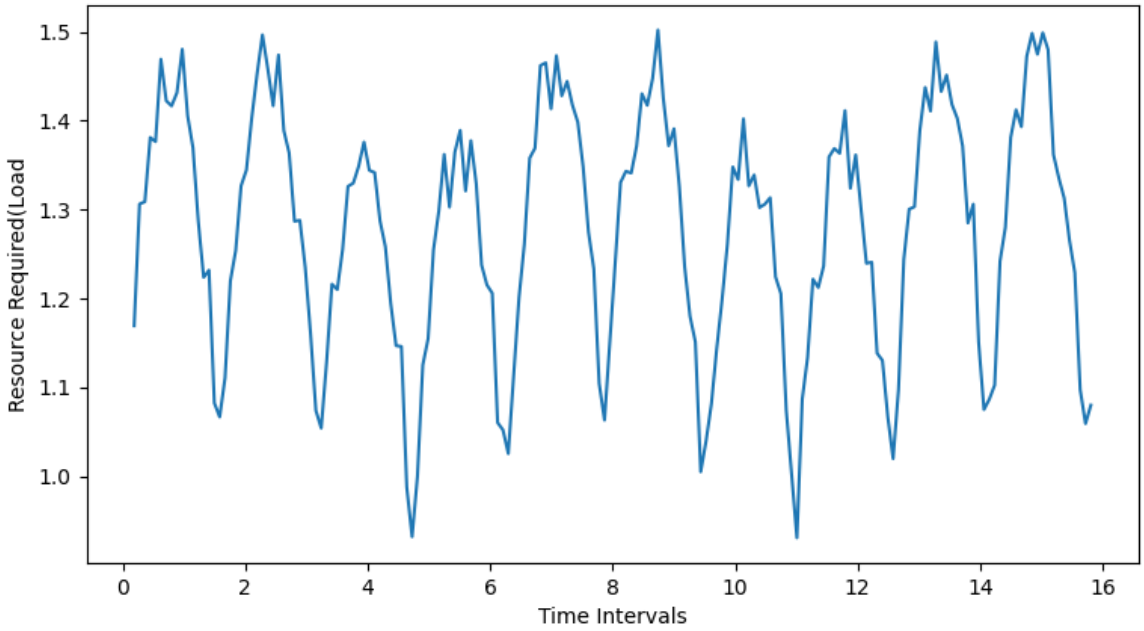


Figure 12: Dataset graphical representation

```
ubuntu@ip-172-31-47-25:/home/autoscaling/x21230064$ python3 sequence.py
ubuntu@ip-172-31-47-25:/home/autoscaling/x21230064$ ls -lrth
total 512K
-rw-rw-r-- 1 ubuntu ubuntu  43K Aug  9 20:03 gru_notebook.ipynb
-rw-rw-r-- 1 ubuntu ubuntu 1.3K Aug  9 20:03 y.csv
-rw-rw-r-- 1 ubuntu ubuntu 9.0K Aug  9 20:03 x.csv
drwxrwxr-x 3 ubuntu ubuntu 4.0K Aug 10 10:54 gru
drwxrwxr-x 3 ubuntu ubuntu 4.0K Aug 10 10:54 cnn_lstm
-rw-rw-r-- 1 ubuntu ubuntu 134K Aug 10 11:41 cnn_lstm.ipynb
-rw-rw-r-- 1 ubuntu ubuntu  11K Aug 10 13:57 ds.csv
-rw-rw-r-- 1 ubuntu ubuntu 1012 Aug 11 15:21 scaler.py
-rw-rw-r-- 1 ubuntu ubuntu 1.4K Aug 11 15:23 node.py
-rw-rw-r-- 1 ubuntu ubuntu  667 Aug 11 15:44 cnn_lstm.py
-rw-rw-r-- 1 ubuntu ubuntu  878 Aug 11 15:44 Execute_CNN_LSTM.py
-rw-rw-r-- 1 ubuntu ubuntu  860 Aug 11 15:44 Execute_GRU.py
-rw-rw-r-- 1 ubuntu ubuntu  892 Aug 11 15:45 Execute_moving_avg.py
-rw-rw-r-- 1 ubuntu ubuntu  886 Aug 11 15:45 Execute_RF.py
-rw-rw-r-- 1 ubuntu ubuntu  882 Aug 11 15:46 Execute_SVR.py
-rw-rw-r-- 1 ubuntu ubuntu  660 Aug 11 15:46 gru_model.py
-rw-rw-r-- 1 ubuntu ubuntu 1.9K Aug 11 15:46 job.py
-rw-rw-r-- 1 ubuntu ubuntu  237 Aug 11 15:47 mean_model.py
-rw-rw-r-- 1 ubuntu ubuntu  547 Aug 11 15:48 rf_model.py
-rw-rw-r-- 1 ubuntu ubuntu  508 Aug 11 18:02 svr_model.py
-rw-rw-r-- 1 ubuntu ubuntu  922 Aug 12 13:17 sequence.py
-rw-rw-r-- 1 ubuntu ubuntu  382 Aug 12 13:18 test.py
-rw-rw-r-- 1 ubuntu ubuntu  27K Aug 12 21:36 cnn_lstm.pkl
-rw-rw-r-- 1 ubuntu ubuntu  27K Aug 12 21:37 gru.pkl
-rw-rw-r-- 1 ubuntu ubuntu  27K Aug 12 21:40 rf.pkl
-rw-rw-r-- 1 ubuntu ubuntu  27K Aug 12 21:40 svr.pkl
-rw-rw-r-- 1 ubuntu ubuntu  27K Aug 12 21:40 avg.pkl
```

Figure 13: Data Generation

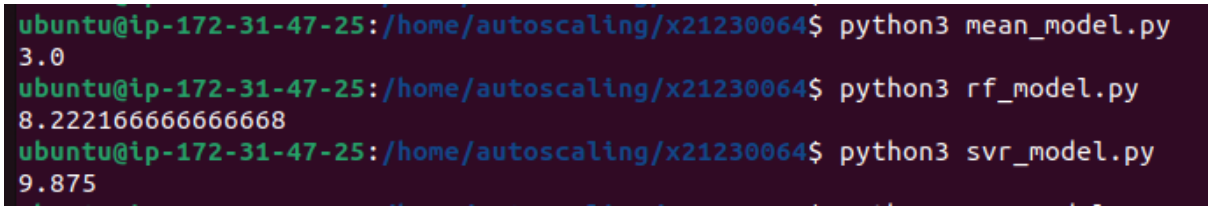


Figure 14: Moving_avg, RF, SVR Model Training

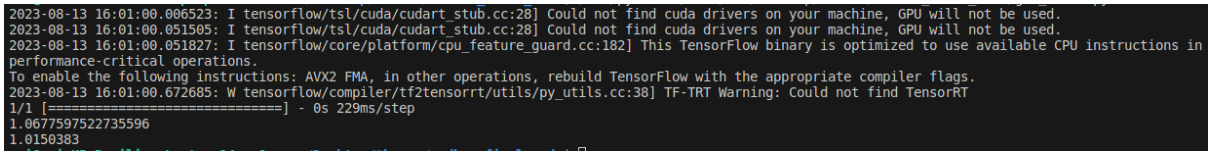


Figure 15: GRU Model Training

- Execute Trained Models on dynamically generated loads: code for this step are included in Execute_Model.py files. Executing these files creates jobs with random loads and the scalar component loads the respective algorithm in the file and predicts the capacity required- the scaling factor to which the resource allocation unit vertically scales the node. visualizer plots graphs for the log data generated by resource allocation unit.

Executing these files creates jobs with random loads and the scalar component loads the respective Model and predicts the capacity required - the scaling factor to which the resource allocation unit scales the node. visualizer plots graphs for the log data generated by resource allocation unit. Figure 17, Figure 18, Figure 8, Figure 20, Figure 21 shows the sample output generated when executing these files for Moving_avg, RF, SVR, GRU, CNN_LSTM models respectivelly.

We Could see in figures that jobs are first loaded and scaler component predicts the required capacity, then the node is scaled to the predicted capacity. Once the task is assigned to the node, if the predicted capacity is less than the required capacity to execute the task, the task has to wait until the node has capacity to execute the job. We can see in the same in sample outputs for few cycles the job executes as soon as it is assigned to node. But for few some cycles, job is delayed for a while.

The visualizer component generates graphs and uploads the same to Amazon S3 bucket. Figure 22 shows the graphs generated and stored in AWS S3 accounts.

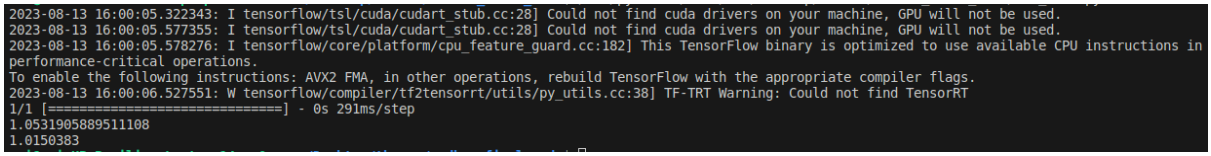


Figure 16: CNN_LSTM Model Training

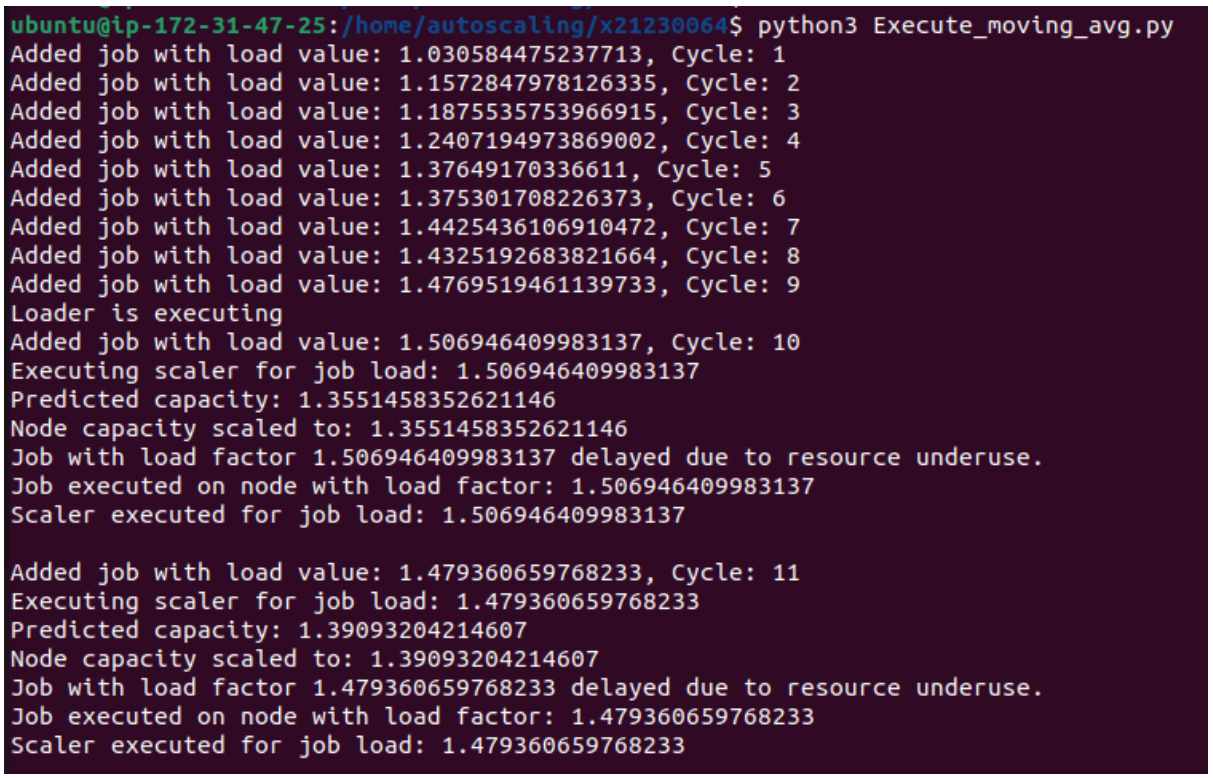


Figure 17: Moving_avg Model Execution on dynamically generated task

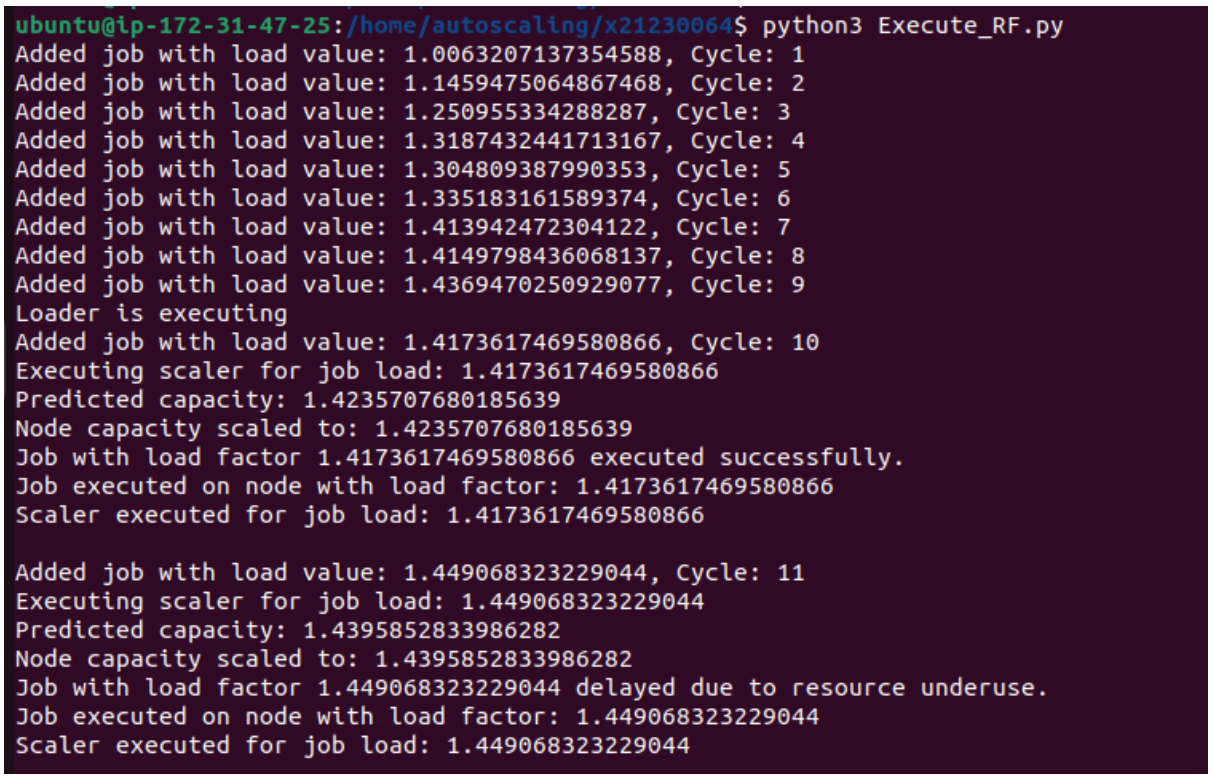


Figure 18: RF Model Execution on dynamically generated task

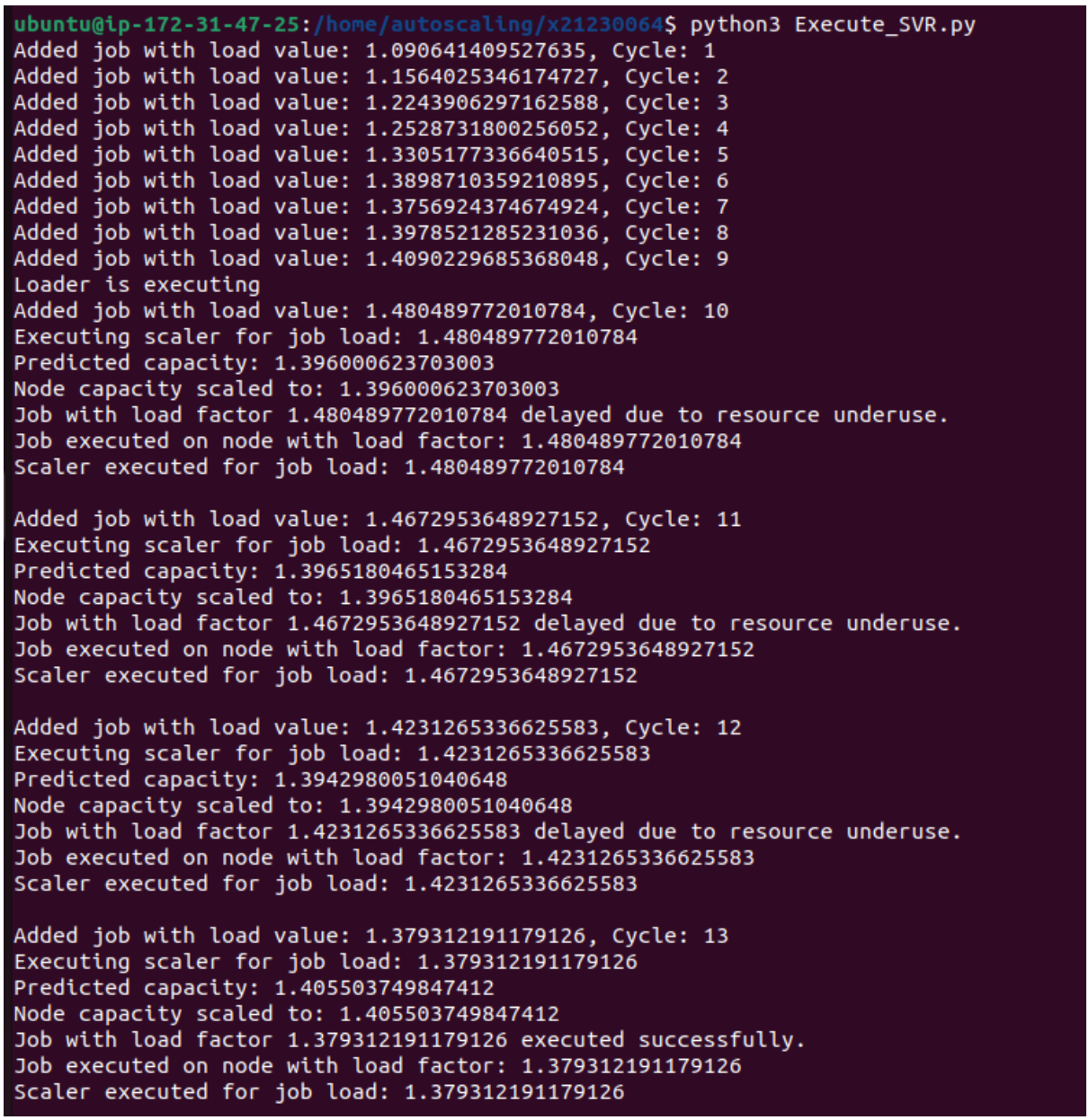


Figure 19: SVR Model Execution on dynamically generated task

```
ubuntu@ip-172-31-47-25:/home/autoscaling/x21230064$ python3 Execute_GRU.py
2023-08-13 15:05:56.892496: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda drivers on your machine, GPU will not be used.
2023-08-13 15:05:56.934962: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda drivers on your machine, GPU will not be used.
2023-08-13 15:05:56.935511: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 AVX512F FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-08-13 15:05:57.591989: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find TensorRT
Added job with load value: 1.024985917056756, Cycle: 1
Added job with load value: 1.1018388620205226, Cycle: 2
Added job with load value: 1.170248231682432, Cycle: 3
Added job with load value: 1.266407754206341 2, Cycle: 4
Added job with load value: 1.359787691068134, Cycle: 5
Added job with load value: 1.4192783214611135, Cycle: 6
Added job with load value: 1.4277463833176602, Cycle: 7
Added job with load value: 1.4682001105016225, Cycle: 8
Added job with load value: 1.4295642162978848, Cycle: 9
Loader is executing
Added job with load value: 1.4466094958102445, Cycle: 10
Executing scaler for job load: 1.4466094958102445
1/1 [==============================] - 0s 268ms/step
Predicted capacity: 1.2998627424240112
Node capacity scaled to: 1.2998627424240112
Job with load factor 1.4466094958102445 delayed due to resource underuse.
Job executed on node with load factor: 1.4466094958102445
Scaler executed for job load: 1.4466094958102445

Added job with load value: 1.4586248409831564, Cycle: 11
Executing scaler for job load: 1.4586248409831564
1/1 [==============================] - 0s 14ms/step
Predicted capacity: 1.3109042644500732
Node capacity scaled to: 1.3109042644500732
Job with load factor 1.4586248409831564 delayed due to resource underuse.
Job executed on node with load factor: 1.4586248409831564
Scaler executed for job load: 1.4586248409831564
```

Figure 20: GRU Model Execution on dynamically generated task

```
ubuntu@ip-172-31-47-25:/home/autoscaling/x21230064$ python3 Execute_CNN_LSTM.py
2023-08-13 15:06:52.575837: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda drivers on your machine, GPU will not be used.
2023-08-13 15:06:52.618903: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda drivers on your machine, GPU will not be used.
2023-08-13 15:06:52.619414: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 AVX512F FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-08-13 15:06:53.270231: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find TensorRT
Added job with load value: 1.0186749131633979, Cycle: 1
Added job with load value: 1.1762775880978076, Cycle: 2
Added job with load value: 1.2492608814842479, Cycle: 3
Added job with load value: 1.3076602905588963, Cycle: 4
Added job with load value: 1.3446671679225919, Cycle: 5
Added job with load value: 1.408660429343624, Cycle: 6
Added job with load value: 1.428285878248388, Cycle: 7
Added job with load value: 1.4913838254389118, Cycle: 8
Added job with load value: 1.4232525626095347, Cycle: 9
Loader is executing
Added job with load value: 1.4662550623645054, Cycle: 10
Executing scaler for job load: 1.4662550623645054
1/1 [==============================] - 0s 336ms/step
Predicted capacity: 1.3218523263931274
Node capacity scaled to: 1.3218523263931274
Job with load factor 1.4662550623645054 delayed due to resource underuse.
Job executed on node with load factor: 1.4662550623645054
Scaler executed for job load: 1.4662550623645054

Added job with load value: 1.4165249216023408, Cycle: 11
Executing scaler for job load: 1.4165249216023408
1/1 [==============================] - 0s 13ms/step
Predicted capacity: 1.3456840515136719
Node capacity scaled to: 1.3456840515136719
Job with load factor 1.4165249216023408 delayed due to resource underuse.
Job executed on node with load factor: 1.4165249216023408
Scaler executed for job load: 1.4165249216023408
```

Figure 21: CNN_LSTM Model Execution on dynamically generated task

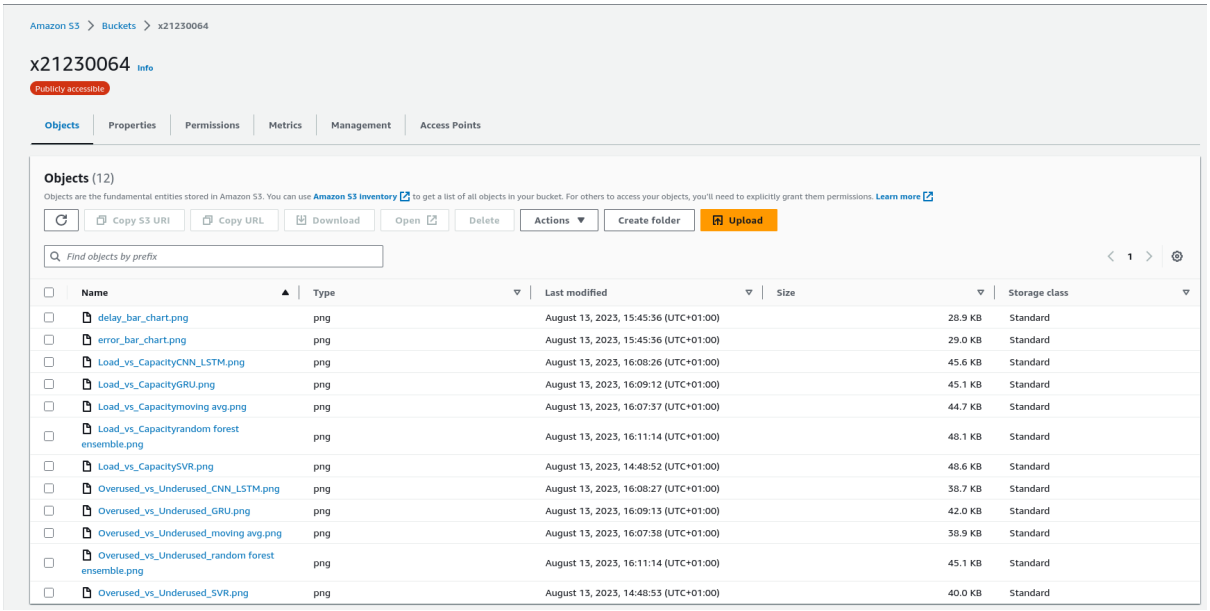


Figure 22: Graphs stored in AWS S3 bucket

## 5.1 Evaluation Metrics

We have used the following evaluation metrics to compare models performance in predicting auto-scaling:

- Overused_vs_Underused: this graph represents whether the predicted capacity for executing the job resulted in over usage or an under utilization of resource.
- Combined error: this is a bar chart representation showing the combined error for different models. Error history from different model instances are retrieved and sum of absolute error values are then plotted.
- Delayed_tasks: This is a graph showing the number of delayed tasks for different models. It retrieves the error history from different model instances and counts the number of delayed tasks (where error ¿ 0) for each model. The number of delayed tasks for different models is then plotted using matplotlib as a bar chart.
- Load vs Capacity: This graph represents the actual load versus predicted load required for executing the task for each model.