# A Prediction Model for Scaling Microservices Oriented Applications

*Student: Deepthi Warrier Edakunni, Advisor: Eyhab Al-Masri*

***Abstract* – With the rapid growth in technology and the proliferation of digital content, Internet users expect web applications to be fast, provide high performance and able to auto scale depending on the number of HTTP web requests on the web application. To provide auto scaling of the applications, application owners depend on the cloud service providers to efficiently auto-scale the web application by horizontal-scaling i.e. adding/ removing instances or by vertical scaling – adding/removing computing resources like CPU or Memory. With the emergence of Internet of Things(IoT), microservice based architectures have become more prevalent & the problem of autoscaling microservices based applications, to provide high performance while reducing the costs has become a challenge. Auto-scaling being very expensive in cloud computing, an IoT developer would end up paying for more resources than needed when all the microservices in the IoT application is auto scaled. For an IoT Application, not all microservices needs to be auto scaled, instead only certain microservices which are overloaded need to be auto scaled. In our case study, we try to solve this problem by building an auto-scaling recommendation system. We ask the user a set of rules and monitor the system based on these rules to auto scale efficiently. Once the threshold is reached for each microservice, the auto scale of that specific microservice is carried out. The results of this recommendation system would identify which microservice to scale, thereby reducing the scaling costs without affecting the overall performance of the application.**

***Keywords — Internet of Things (IoT), Cloud, Microservices, Auto-scale, Kubernetes, load-balancing, Predictive autoscaling.***

## INTRODUCTION

Due to the emerging cloud computing paradigm and its most appealing feature of elasticity, web application owners have been migrating their applications to public cloud providers like Amazon Web Services(AWS) & Microsoft Azure, Google Cloud Platform. This allows the cloud users to "horizontally" scale-out to add and scale in to remove instances to maintain performance or to reduce costs. It is also possible to do vertical scaling by reconfiguring VMs by adding (scale-up) or removing (scale-down) computing resources.

In the case of a multi-tier monolithic architecture style of web applications, an application instance of the user presentation layer and the business logic tier is generally hosted by a VM. Since most of the application logic and processes happen on this tier, it is relatively easy to replicate identical instances of this tier on demand allowing the web application to grow horizontally.

But with the rapid growth of Internet of Things, the microservice based architecture is becoming more prevalent. An IoT application consists of tens or hundreds of microservices. These microservices can be full stack applications with a dedicated data persistence tier or can be stateless and would be working together in real time binding. If these microservices do not adapt to load changes, the entire IoT application's performance would be degraded and is significantly impacted.

The solution to this problem would be to approach cloud providers to auto scale the resources. However, since an IoT application is composed of several microservices, autoscaling of all the microservices would lead to an IoT developer end up paying for resources that are not necessary. So, the challenge with the microservice based architecture applications is to

identify “Which microservice should be scaled?” This work focuses on the development of a solution to identify which microservice to scale and thereby allowing autoscaling of microservices architectures in an efficient way.

## RESEARCH PROBLEMS

This case study will investigate on some of the important research questions for proposing a prediction model for autoscaling microservice based IoT Applications. To this extent, we identify the following research questions that we plan to address.

**RQ1: Can we identify one microservice to scale up in an IoT Application and not the others?**
In the case of microservice based IoT Applications, there will be tens or hundreds of microservices that would make up the Application. All these microservices would not be loaded with requests at the same time. So, our goal would be to identify which microservices are overloaded with requests and which of them needs to be auto scaled. We will be simulating this process, by using a client tool like Locust.io tool, an open source load testing tool[1] to generate request for one of the microservices there by just biasing the traffic to the specific microservice.

**RQ2: What constraints will be used to identify if the workload on the microservice is high?**
Threshold-based auto-scaling methods are generally the standards that are used by the cloud providers. This would include monitoring the CPU Usage or Memory Utilization, the request rate for each microservice. The application owner sets an arbitrary threshold values for the above-mentioned parameters and a python script would-be run-in background to identify if that threshold has reached for each microservice. Based on these values, a decision is taken if the individual microservice must be scaled or not.

**RQ3: Are there features offered by public cloud providers like AWS, Microsoft Azure and Google Cloud Platform to independently scale different micro services ? What is the architecture pattern to accomplish this?**

Most of the cloud providers are now migrating to support microservices based applications due to its obvious benefits and advantages.

The research done by Amazon Web Services as mentioned in [3], a traditional monolithic architecture is hard to scale. If one feature in the application experiences a spike in demand, the entire application must be scaled. [3] mentions about decoupling the monolithic application into microservices. Each microservice is run as a separate service within its own container and hence can be scaled independently.

Microsoft as well provides an option for deploying microservices to Azure Kubernetes Service (AKS)[5]. Similarly, Google provides an option to deploy microservices on Google Kubernetes Engine (GKE)[6].

## METHODOLOGY

For the scope of the research, we limit to use Amazon Web Services Platform to build, deploy, and test the project. There are two methodologies that can be followed for the data collection on AWS, though we have followed Approach2 for intensive study and research purposes.

## APPROACH 1

The methodology that will be followed to deploy the microservices architecture is to have each application component run as its own service within its own container and each service communicates with the other services via a well-defined API. As previously mentioned in the Introduction, microservices are built around business capabilities and each service performs independently as a single function.

The research uses Amazon AWS platform to deploy, test, and collect the metrics like the CPU Usage, Memory Utilization, Network Throughput for each of the microservice. Docker build is used to package each of the microservices code along with its libraries, dependencies and build the service using docker commands to create an image. Once the build is successful, docker tag command is used to tag the image and

then pushed to the Amazon Elastic Container Registry(ECR). This process is done for each of the microservices, so that each microservice will have its own individual image and the images are pushed into the ECR independently. The next step is to deploy these images to the Amazon Elastic Container Service(ECS).

An Application Load balancer(ALB) is used to route the request to the individual Target Groups which keeps track of the individual instance and ports for each container running that service[6]. These microservices can then be individually auto scaled based on the metrics – CPU Usage, Memory Utilization and Network Throughput.

The scope for the research is to identify the above-mentioned metrics for each of the microservices and to log the results onto S3 or an RDS Instance. The results are then analyzed, and a decision is made on when to auto scale which microservice. The decision to auto scale could be based on an individual metric value or a combination of two or three metric values. For example, a combination of values for CPU Usage, Memory Utilization and the incoming requests would be an appropriate metric that could be used to identify if a microservice needs to be auto scaled i.e. scale out by adding new instances or scale in by removing instances, thereby improving performance or reducing the costs respectively.

To identify the metric values, a script will be written for each microservice to capture the individual metric values. The script will be packaged as part of the docker image for each microservice when pushed to the ECR and thereafter deployed onto the ECS. A parent script is scheduled to call the individual microservice scripts every 10 minutes, thus ensuring the data collected from each microservice is time synchronized. The individual microservice scripts are also responsible to write the collected information onto the database/file.

The scope of the research is to focus on collecting the following information from each microservice – CPU Usage, Memory Utilization, Network Throughput. The above-mentioned metrics for each microservice are then recorded onto a database every 10 minutes.

## ARCHITECTURE DIAGRAM

The architecture diagram for the methodology explained in the previous section is as below:

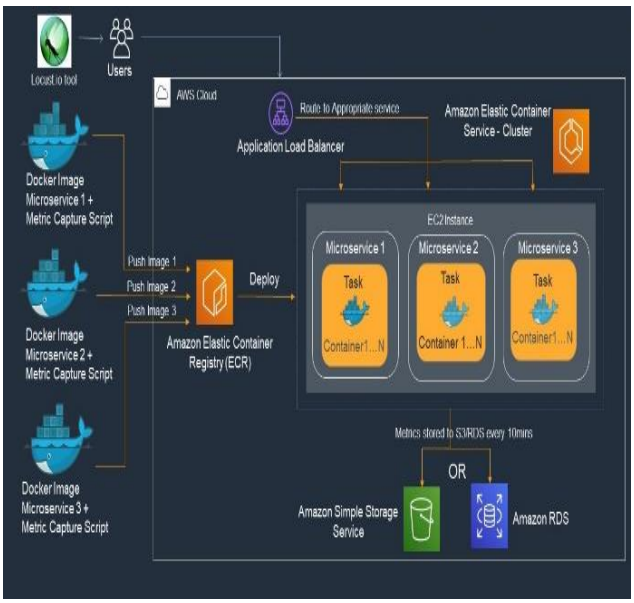


*Figure. 1[8]*

## DATA COLLECTION STEPS

The prerequisite to the collection of the metrics for the docker containers is to deploy the microservices to the ECS Cluster. The steps to be followed for the deployment is as mentioned in the above methodology section.

Below are the steps followed for the data collection from the docker containers.

1. Cloud Formation:
   Created a cloud formation template with changes to the specific parameters in the ecs.yml file.
   a) Key/Pair: The KeyName parameter for keypair to enable SSH/Putty(Windows OS)access to the EC2 instances which will host the docker containers.
   b) Installing AWS CLI: The command to install aws cli is given as part of

the UserData Section of the cloud formation template ecs.yml
*sudo yum install -y aws-cli*

*c)* Full Access to S3: Full access to read/write files to the S3 bucket was provided as part of the EC2 Role for the IAM Role Type in the Policies-Action attribute: - *'s3:*'*.

2. SSH/Connect via putty to the EC2Instance:
   a. To login, the SSH Authentication needs to be done by providing the authentication.pem file.
3. Check for the aws cli installation in the EC2Instance.
   Type in the below command:
   *aws –version*
4. Collect the metrics:
   The metrics that are collected for the containers are Container Id, Name, CPU %, Mem Usage / Limit, Mem %, Net I/O, Block I/O and Pids. The metrics are collected every minute and appended to the same csv file for each day. The docker stats command is used to collect the metrics. The command for collecting the metrics is as given below:
   *while true; do docker stats --no-stream –format "table {{.ID}},{{.Name}},{{.CPUPerc}},{{.MemUsage}},{{.MemPerc}},{{.NetIO}},{{.BlockIO}},{{.PIDs}}" | cat >> ./`date -u +"%Y%m%d.csv"`; sleep 60; done*
5. Transfer the csv file to the S3 bucket.
   The command to copy the generated file to the S3bucket is as below:
   *aws s3 cp ./`date -u +"%Y%m%d.csv"` s3://tcss600/dockerstats/`date -u +"%Y%m%d.csv"`;*

## FLOWCHARTS

The flowchart for the microservices deployment is as below:

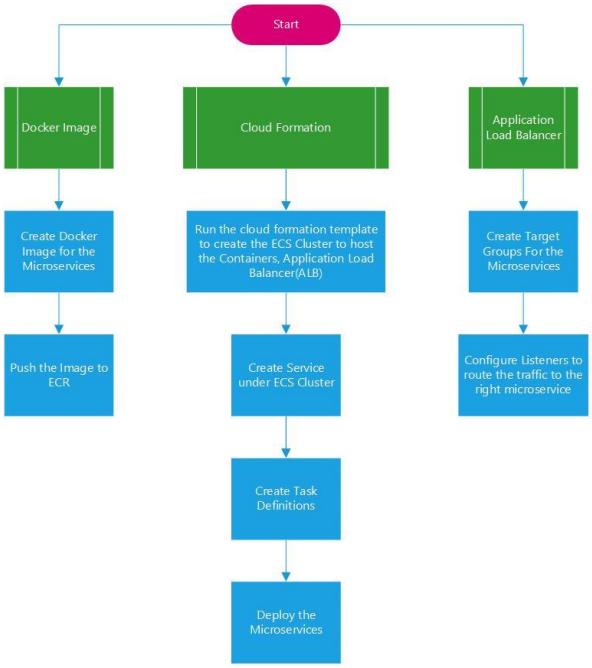


*Figure 2. Deployment of Microservices*

The flowchart for the data collection is as below.

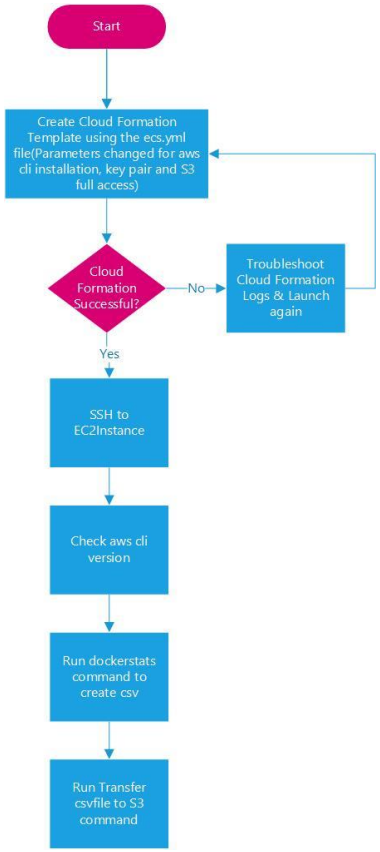


*Figure 3. Data Collection Flowchart*

## APPROACH 2

This methodology of deploying the microservices to Kubernetes was used to conduct intensive study and for research. Kubernetes being an open source platform for managing containerized services and with its powerful feature like portability & extensibility, makes it a popular platform for managing containers.

***Local Development:***

For the development and learning purpose, the tool - Minikube is used to run Kubernetes locally. Kubectl is a simple CLI tool that is used to interact with the cluster created by Minikube. A yaml file – Deployment.yaml is used to deploy the services onto the cluster. The deployment.yaml file holds the details for how many deployments and the number of replicas that must be created and the service details .

***Cloud Platform & Tools required:***

***Docker:***
The first step towards deployment is to create individual docker images for the microservices. Docker build is used to package each of the microservices code along with its libraries, dependencies and build the service using docker commands to create an image. Once the build is successful, the image is pushed to the Docker Hub.

***Cloud Platform – AWS EKS:***
For research purposes, to collect the metrics, the docker images needs to be deployed to a cloud platform. For this study, Amazon Elastic Kubernetes Service(EKS) platform is used to deploy the micro services.

***Eksctl :***
Eksctl is a simple CLI Tool very similar to the kubectl but has been custom developed by for Amazon in collaboration with Weave works. Eksctl is used for creating clusters on the Amazon EKS platform and for interacting with the Amazon clusters.

***Data source - Prometheus:***
Prometheus is an open source data monitoring tool which helps in time series collection of data via the pull model over HTTP.

***Visualization – Grafana:***
For visualization of the data collected by Prometheus, an open source analytics and visualization software called Grafana is used. It allows to query the data source using PromQL Queries and visualize the data, create graphs, and explore the metrics collected by the monitoring tool. It also provides an option to export the data in an easy format like csv file which would later be used for analysis and prediction.

***Helm:***
A package manager and application manager tool for managing installations and upgrades for Kubernetes.

***Locust.io:***
Locust is an open source load testing tool used to simulate several simultaneous users to a specific api. For research and testing purposes, one of the microservices is swarmed with a greater number of requests compared to the others and the metrics – CPU Usage, Memory Usage and Network I/O is collected for the over loaded microservice [9].

***Jupyter Notebook:***
Jupyter Notebook is an open-source web application that is mainly used for statistical modeling , machine learning, data visualization and for creating prediction models.

## ARCHITECTURE DIAGRAM

The Architecture diagram for the Kubernetes System is as below:

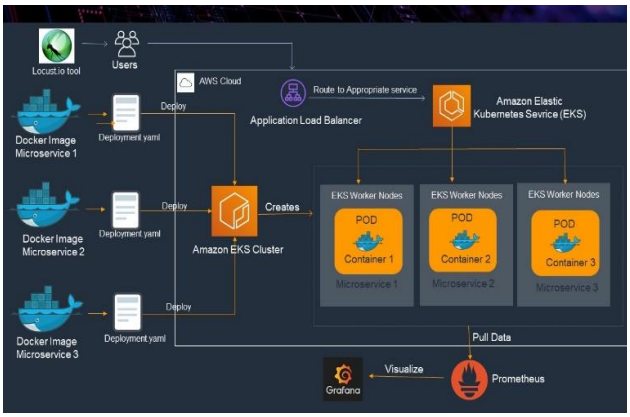


*Figure 4. Microservices on Amazon EKS Architecture diagram*

## DATA COLLECTION STEPS

Below are the steps followed for the data collection on Amazon EKS.

1. Create docker images for the microservices using docker commands. Push the images to Docker Hub.
2. Launch and configure the EKS Cluster and worker nodes using the eksctl CLI tool.
3. Deploy the Kubernetes Dashboard.
4. Use the token to login to the Kubernetes Dashboard and ensure that the dashboard is accessible.
5. Mention the image details for the microservice, deployment including the number of replicas and service details in the corresponding deployment.yaml file and deploy the microservice to the cluster. Kubernetes will ensure the preferences mentioned in the yaml file is honored when the application is deployed.
6. Metric Server api needs to be installed on to the Amazon EKS cluster for collecting the resource usage data.
7. Install Prometheus to monitor the metrics – CPU Usage, Memory Usage, Network I/O for the Pods, and the containers and store the data in gp2 EBS Volumes.
8. Use Grafana and connect to the data source - Prometheus for visualizing the metrics collected using PromQL Queries.
9. Setup Locust tool to swarm one of the microservices (tcss600users) with requests.

## EXPERIMENTS & RESULTS

The microservices are deployed onto Amazon EKS as tcss600users, tcss600posts, tcss600threads. 3 replicas/pods are created for each of the deployments using the corresponding yaml files. Once all the microservices are deployed properly, ensure the deployments were successful by accessing the corresponding URLs and checking the Kubernetes dashboard. Kubernetes dashboard would display all the deployments and the corresponding pods as shown below in figure 5.

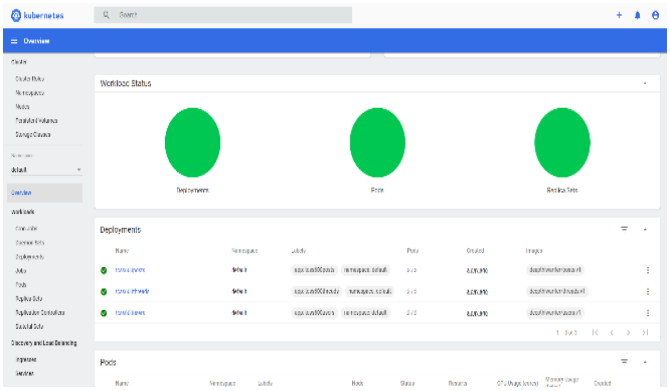


*Figure 5. Kubernetes Dashboard*

Locust.io is used to swarm one of the microservices(tcss600users) with 500 requests per second and the CPU Usage, Memory Usage and Network I/O is collected for the tcss600users deployment.

The screenshots for the Locust I/O is as below in figure 6 and 7.

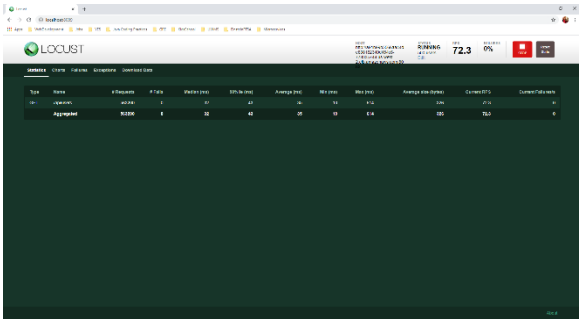


*Figure6. Locust.io Dashboard*

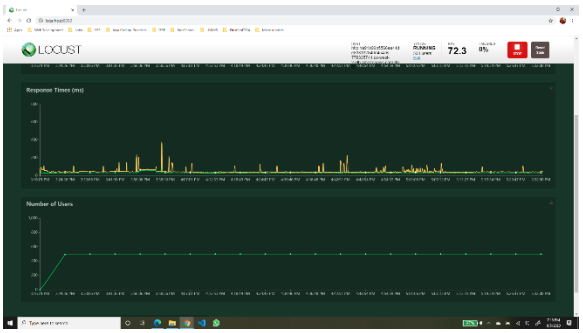


*Figure 7. Locust.io users and response time dashboard*

Prometheus is used to collect the metrics - CPU Usage, Memory Usage and Network I/O for all the pods that are deployed. The metrics are collected for two days by Prometheus is then visualized using Grafana.

Few sample screenshots of the Grafana dashboard is as shown below in figure 8 & figure 9.

*Figure 8. Grafana Dashboard*

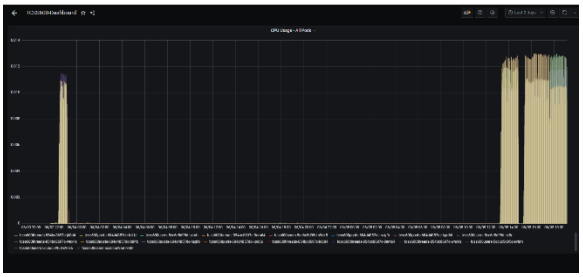

*Figure 9. Grafana Dashboard – tcss600users - CPU Usage*

The data visualized by Grafana is then exported to a csv file for further analysis and prediction. Jupyter notebook is used for the analysis's purposes due to its wide range of support for data science tools.

The 'pandas' package is used to read the csv files of the datasets that were exported from Grafana. All the metrics that have zero value for Memory is then removed, treating them as missing data and hence to exclude them from further analyses.

The correlation between the metrics and the status is identified and it indicates a positive relationship. This means that as the Network I/O increases, the CPU Usage and the Memory Usage also increases, and the status also changes its value from 0 to 1.

The model predicts the value of status based on the combined value of CPU Usage, Memory Usage, and the Network I/O. The status of zero indicates that the container is not overloaded and hence no auto scaling measures needs to be taken. A status of 1 indicates that autoscaling will have to be done on the overloaded containers.

To visualize this relationship, a scatter plot is drawn, and a linear regression model used for understanding this relationship. The packages – statsmodels and the scikit-learn and the built-in methods are used for developing the predictive model. The statsmodels package has an OLS class that can be used to run linear regression models.

The following code will create a linear regression model with status as the outcome with a combination of values from CPU, Memory, and the Network I/O. The fit() method will the model to the data given as input.

```
ols('Status ~ CPU + Memory + Network_IO',metrics_user_pod1_var_df).fit()
```

The summary() method returns the OLS Regression Results. The OLS Regression results for the tcss600users deployment is shown in figure 10.

OLS Regression Results

| | | | |
|---|---|---|---|
| Dep. Variable: | Status | R-squared: | 0.899 |
| Model: | OLS | Adj. R-squared: | 0.899 |
| Method: | Least Squares | F-statistic: | 4010. |
| Date: | Mon, 08 Jun 2020 | Prob (F-statistic): | 0.00 |
| Time: | 19:34:54 | Log-Likelihood: | 1034.6 |
| No. Observations: | 1358 | AIC: | -2061. |
| Df Residuals: | 1354 | BIC: | -2040. |
| Df Model: | 3 | | |
| Covariance Type: | nonrobust | | |

| | coef | std err | t | P>\|t\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| Intercept | 0.0113 | 0.004 | 3.169 | 0.002 | 0.004 | 0.018 |
| CPU | 236.4056 | 33.769 | 7.001 | 0.000 | 170.161 | 302.650 |
| Memory | 2.493e-07 | 9.38e-08 | 2.658 | 0.008 | 6.53e-08 | 4.33e-07 |
| Network_IO | -0.0002 | 4.63e-05 | -4.359 | 0.000 | -0.000 | -0.000 |

| | | | |
|---|---|---|---|
| Omnibus: | 1785.806 | Durbin-Watson: | 0.191 |
| Prob(Omnibus): | 0.000 | Jarque-Bera (JB): | 198254.533 |
| Skew: | 7.377 | Prob(JB): | 0.00 |
| Kurtosis: | 60.324 | Cond. No. | 5.79e+08 |

*Figure 10. OLS Regression Results*

With the model now fit to the data, the predict() method is now used to plot the regression line. CPU Usage vs. Predicted Status can be plotted as in figure 11.

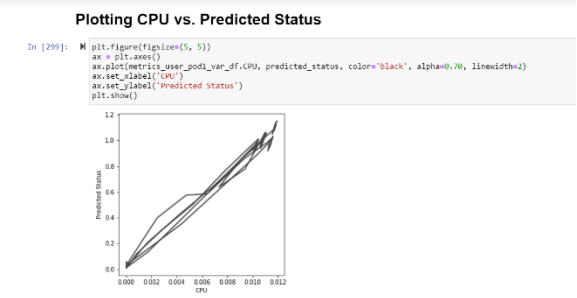


*Figure 11. CPU vs Predicted Status*

The plot for the actual result vs. the predicted results determines if the prediction model would work. The plot is as shown below.

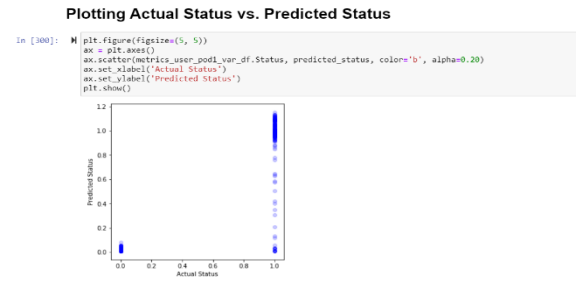


*Figure 12. Plotting Actual Status vs. Predicted Status*

Based on the results predicted for the status, it would be useful for the developers to identify when the containers are overloaded, and decision can be made if the services should be scaled up/ scaled down.

***Problems/ shortcomings of the study:***

Since the deployment was done on Amazon EKS, the cost associated with maintaining the EKS cluster and the associated EC2 Instance was very high. The free tier credits were not enough to run the experiment for a longer period. Doing an extensive study for the prediction model is an expensive option for students. This impacted the number of datasets collected for extensive study.

## CONCLUSION & FUTURE WORK

This paper summarizes the methodologies which can be adopted to predict auto scaling of microservices based IoT Application Architectures. The focus was on Kubernetes based approach on Amazon Web Services in this study. Based on the metrics collected – CPU Usage, Memory Usage and Network I/O and a machine learning model, a prediction is made regarding the status. Using this prediction, IoT developers can make an informed decision for autoscaling.

***Future directions:***

Some of the features that could be based upon the model will be:

1. The prediction made by the model needs to be acted upon to auto scale (up/down) by using a script.
2. Monitor the metrics for an extended period and based on the observations, predict an auto scale for a future point of time. For example, say if a specific container is overloaded on a specific day of the week at a certain time, the analysis and the model would help in predicting an auto scale of the containers for a future day at the same time as analyzed.
3. Running the project on cloud platforms other than AWS would also help in comprehensive study.

## REFERENCES


[1] N. Cruz Coulson, S. Sotiriadis and N. Bessis, "Adaptive Microservice Scaling for Elastic Applications," in IEEE Internet of Things Journal, vol. 7, no. 5, pp. 4195-4202, May 2020, doi: 10.1109/JIOT.2020.2964405.

[2] Break a Monolith Application into Microservices with Amazon Elastic Container Service, Docker, and Amazon EC2, "https://aws.amazon.com/getting-started/hands-on/break-monolith-app-microservices-ecs-docker-ec2/"

[3] Implementing Microservices on AWS, “https://d1.awsstatic.com/whitepapers/microservices-on-aws.pdf”

[4] Microservices architecture on Azure Kubernetes Service (AKS), “https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/microservices/aks”

[5] Migrating a monolithic application to microservices on Google Kubernetes Engine, “https://cloud.google.com/solutions/migrating-a-monolithic-app-to-microservices-gke”

[6] Satish Narayana Srirama, Mainak Adhikari, Souvik Paul, “Application deployment using containers with auto-scaling for microservices in cloud environment” in ScienceDirect Volume 160, 15 June 2020, 102629

[7] What is Locust? “https://docs.locust.io/en/stable/what-is-locust.html”

[8] Quick Start Program with Locust, “https://docs.locust.io/en/stable/quickstart.html”

[9] Figure1 Icon Templates, “https://aws.amazon.com/architecture/icons/”

[10] What is Kubernetes? , “https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/”

[11] Amazon EKS Workshop, “https://eksworkshop.com/”

[12] Prometheus, “https://prometheus.io/docs/introduction/overview/”

[13] Grafana, “https://grafana.com/docs/grafana/latest/getting-started/what-is-grafana/”

[14] Installing the Kubernetes Metrics Server, “https://docs.aws.amazon.com/eks/latest/userguide/metrics-server.html”